SIDnet-SWANS v1.0 Manual

Oliviu C. Ghica - Northwestern University

July 30, 2008

Valid for SIDnet-SWANS v.1.2.0 and later.

Contents

1 LICENSE

Copyright (c) 2008 Northwestern University, Inc & Oliviu C. Ghica. All rights reserved. Authors: Oliviu C. Ghica

The following information refers exclusively to the SIDnet-SWANS software package, excluding the content of ”importedpackages” (JiST-SWANSv1.0.6) where separate, specific licenses apply. SIDnet-SWANS uses the JiST-SWANSv1.0.6 for non-commercial purposes.

This software is licensed by Northwestern University for non-commercial academic purposes only. By using this software, you hereby enter into the following licensing agreement with Northwestern University.

Legal Notice concerning the following included package (since SIDnet-SWANS version 1.1.0) $sidnet.mac.mac802_15_4$ /* * Copyright (c) 2003-2004 Samsung Advanced Institute of Technology and * The City University of New York. All rights reserved. * * Redistribution and use in source and binary forms, with or without * modification, are permitted provided that the following conditions * are met: * 1. Redistributions of source code must retain the above copyright * notice, this list of conditions and the following disclaimer. * 2. Redistributions in binary form must reproduce the above copyright * notice, this list of conditions and the following disclaimer in the * documentation and/or other materials provided with the distribution. * 3. All advertising materials mentioning features or use of this software * must display the following acknowledgement: * This product includes software developed by the Joint Lab of Samsung * Advanced Institute of Technology and The City University of New York. * 4. Neither the name of Samsung Advanced Institute of Technology nor of * The City University of New York may be used to endorse or promote * products derived from this software without specific prior written * permission. * * THIS SOFTWARE IS PROVIDED BY THE JOINT LAB OF SAMSUNG ADVANCED INSTITUTE * OF TECHNOLOGY AND THE CITY UNIVERSITY OF NEW YORK “AS IS” AND ANY EXPRESS * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN * NO EVENT SHALL SAMSUNG ADVANCED INSTITUTE OR THE CITY UNIVERSITY OF NEW YORK * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. */ End of Legal Notice concerning the following included package (since SIDnet-SWANS version 1.1.0) $sidnet.mac.mac802_15_4$

Software License Terms and Conditions

1. SOFTWARE shall mean SIDnet code, or portions thereof, or related

documentation, made available on the project web site. SOFTWARE includes, but is not limited to, SIDnet source code, object code and executable code. SOFTWARE excludes the content of
importedpackages (JiST-SWANSv.1.0.6) where separate licenses apply.

2. Northwestern University and Oliviu C. Ghica holds all intellectual property rights in SOFTWARE, including but not limited to copyright, trademark and patent rights.

3. LICENSEE means the party to this Agreement and the user of SOFTWARE. By using SOFTWARE, LICENSEE enters into this Agreement with Northwestern University.

4. SOFTWARE is made available under this Agreement to allow certain non-commercial academic use. Northwestern University reserves all commercial and non-academic rights to SOFTWARE and these rights may be licensed by Northwestern University to third parties. License for the "importepackages" (JiST-SWANS distribution) must be also obtained from CRF and Cornell University according to JiST-SWANSv1.0.6 separate licenses)

5. LICENSEE is hereby granted permission to download, compile, execute, copy, and modify SOFTWARE for non-commercial academic purposes provided that this notice accompanies all copies of SOFTWARE. Copies of modified SOFTWARE may be distributed only for non-commercial academic purposes (a) if this notice accompanies those copies, (b) if said copies carry prominent notices stating that SOFTWARE has been changed, and (c) the date of any changes are clearly identified in SOFTWARE.

6. LICENSEE agrees that the export of SOFTWARE from the United States may require approval from the U.S. government and failure to obtain such approval will result in the immediate termination of this license and may result in criminal liability under U.S. laws.

7. Northwestern University provides SOFTWARE on an "as is" basis. Northwestern University does not warrant, guarantee, or make any representations regarding the use or results of SOFTWARE with respect to its correctness, accuracy, reliability or performance. The entire risk of the use and performance of SOFTWARE is assumed by LICENSEE. ALL WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR MERCHANTABILITY AND ANY WARRANTY OF NONINFRINGEMENT OF PATENTS, COPYRIGHTS, OR ANY OTHER INTELLECTUAL PROPERTY RIGHT ARE HEREBY EXCLUDED.

8. LICENSEE understands and agrees that neither Northwestern University nor Oliviu C. Ghica is under any obligation to provide maintenance, support or update services, notices of latent defects, correction of defects, or future versions for SOFTWARE.

9. Even if advised of the possibility of damages, under no circumstances shall Northwestern University and Oliviu C. Ghica individually or jointly be liable to LICENSEE or any third party for damages of any character, including, without limitation, direct, indirect, incidental, consequential or special damages, loss of profits, loss of use, loss of goodwill, computer failure or malfunction. LICENSEE agrees to indemnify and hold harmless Northwestern University for any and all

liability Northwestern University may incur as a result of use of SOFTWARE by LICENSEE.

2 Installation Instructions

2.1 Getting the Software

1. Java Integrated Development Environment We recommend, in order, the following two, most popular java IDEs (* following instructions will be made with respect to NetBeans 5.5.1 IDE):

   - NetBeansIDE (”http://www.netbeans.org/”)
   - EclipseIDE (”http://www.eclipse.org/”)

2. Java Development Kit 5.x (6.x has not been tested yet) ”http://java.sun.com/javase/downloads/index.jsp”

3. SIDnet-SWANS distribution package (reading this assumes that you have already done so)

The kit includes the JiST/SWANS components.
Un-archive the contents of the files in a directory of your choice.

2.2 Setting up a Project

The following instructions refer to NetBeansIDE 5.5.1. These should be somehow analogous to other IDE environments.

2.2.1 Establish a SIDnet-SWANS project

1. Open NetBeansIDE

2. File → New Project

   (a) Categories: General
   (b) Projects: Java Project with Existing Sources
   (c) Choose a project name (e.q. ”SIDnet”)
   (d) Indicate the project folder (it should be /SIDnet-SWANS - the installation directory)
   (e) Click Finish

The Project will appear on the left panel. Under the ”Files” tab, you may browse through its files.

2.2.2 Configure the project dependencies

1. Right-click on the project icon in the left panel, and click Properties
2. Under Categories frame, click Sources
   (a) Make sure that the Source Level is 1.5 (JDK 5). Newer source level might work, but it has never been tested for.
   (b) Under Source Package Folders:
      i. Add Folder : "SIDnet-SWANS/importedpackages/jist-swans-1.0.6/src"
      ii. Add Folder : "SIDnet-SWANS/src"
      iii. Add Folder - : "SIDnet-SWANS/libs/opencsv-1.8/src"
3. Under Categories frame, click Libraries
4. Add the following list of JAR/Folders (libraries)
   (a) SIDnet-SWANS/libs/BCEL/org/apache/bcel-5.2/bcel-5.2.jar
   (b) SIDnet-SWANS/libs/jaxb-ri-20070917/lib/jaxb-api.jar
   (c) SIDnet-SWANS/libs/jaxb-ri-20070917/lib/jaxb-impl.jar
   (d) SIDnet-SWANS/libs/apache-log4j-1.2.15/log4j-1.2.15.jar
   (e) SIDnet-SWANS/importedpackages/jist-swans-1.0.6/libs/bsh.jar
   (f) SIDnet-SWANS/importedpackages/jist-swans-1.0.6/libs/checkstyle-all.jar
   (g) SIDnet-SWANS/importedpackages/jist-swans-1.0.6/libs/jargs.jar
   (h) SIDnet-SWANS/importedpackages/jist-swans-1.0.6/libs/jython.jar
5. Make sure that the Java Platform is selected as JDK 1.5
6. Add the following Library: *Swing Layout Extensions*
7. Under Categories frame, click Run
   (a) Main Class: "jist.runtime.Main"
   (b) Arguments: "jist.swans.Main sidnet.stack.driver.Driver_SampleP2P 300 4000 1000000"
   (c) Working Directory: "SIDnet-SWANS/"
   (d) Click OK
8. Do a complete build: Menu→Build→Build Main Project

2.3 Run SIDnet-SWANS application

2.3.1 From command line

Assuming the you have previously built the entire package:

java jist.runtime.Main jist.swans.Main sidnet.stack.driver.Driver_Name N# L# T#

where:

- N#: number of nodes
- L#: width [ft] of the simulation field (assumes square field)
- T#: simulation time constant (long-valued) at which the simulator to stop automatically

For example:

java jist.runtime.Main jist.swans.Main sidnet.stack.driver.Driver_kShortestPath 300 4000 10000000

2.3.2 From within NetBeans 5.5.1

1. Compile: From Menu Build → Compile File (F9)
2. Build : From Menu Build → Build Main Project (F11)
3. Run : From Menu Run → Run Main Project (F6)

If everything was set-up correctly, the GUI-window should show up on the screen.

If you encountered problems, make sure you have followed this steps precisely. You may also consider the Troubleshooting section.

3 Sample Application ("Hello World")

This chapter will walk you through the set-up, execution and run-time interaction with a "hello world"-like application.

The scenario is as follows: N wireless sensor nodes are being randomly deployed in a square area of length L. The simulation will self-terminate after T seconds. A user walks through this area, connects to one of the nodes through a terminal (i.e., laptop, PDA) and submits a simple query. Through the query, the user asks for measurements of a particular phenomenon from a particular sub-region. The user is interested in achieving 60 samples over an one hour interval.

To build this application, we need the following three files

1. Application Layer implementation (sidnet/stack/app/AppSampleP2P.java)
2. Routing algorithm (sidnet/stack/routing/ShortestGeographicalPathRouting.java)
3. Driver (sidnet/stack/driver/Driver_SampleP2P.java)

The application layer implements the barebones for this scenario to work: awaits for user interaction, sends the query request and processes an incoming query by sampling the underlying phenomenon and submitting the measurements back to the user. The node that the user connects and submits the query through becomes the *sink* node. The node that samples the data becomes the *source* node.

The routing algorithm implements the path-construction between the sink and the source nodes. At a minimum, this can be achieved through a P2P-type protocol. If we assume (and we do for this scenario) that location-information is available (i.e, GPS equipped nodes) then the algorithm is quite simple: every node on the "path" forwards the data-packet to the node that is closest to the sink. Ultimately, the data-packets will reach the sink. We won't cover exceptions for this sample application.

The "driver" binds all the information together and represents the entry point in a SIDnet application. In a driver you specify the network stack of *each* sensor node, the implementing algorithms at each layer, the phenomena that will be measured, the placement of the nodes in the area of interest, etc.

To run this application, type the following at the command line

java java.runtime.Main java.swans.Main sidnet.stack.driver.Driver_SampleP2P 500 5000 100000

($N = 500$, $L = 5000$, $T = 100000$)

Simply put, the SIDnet driver runs on top of SWANS, which runs on top of JiST engine, which in turn runs on top of the JVM (Java Virtual Machine).

The GUI should pop-up. Initially, there will be an one-hour "boot" period in which nodes discover their neighboring nodes. We have programmatical accelerated this section of code. After one-hour, simulator slows down to quasi-real time awaiting for user's interaction.

Pick up a node of your choice, right-click on it and select "Connect Terminal to ...". A *terminal* window will show up just like in Figure 1. It will show you

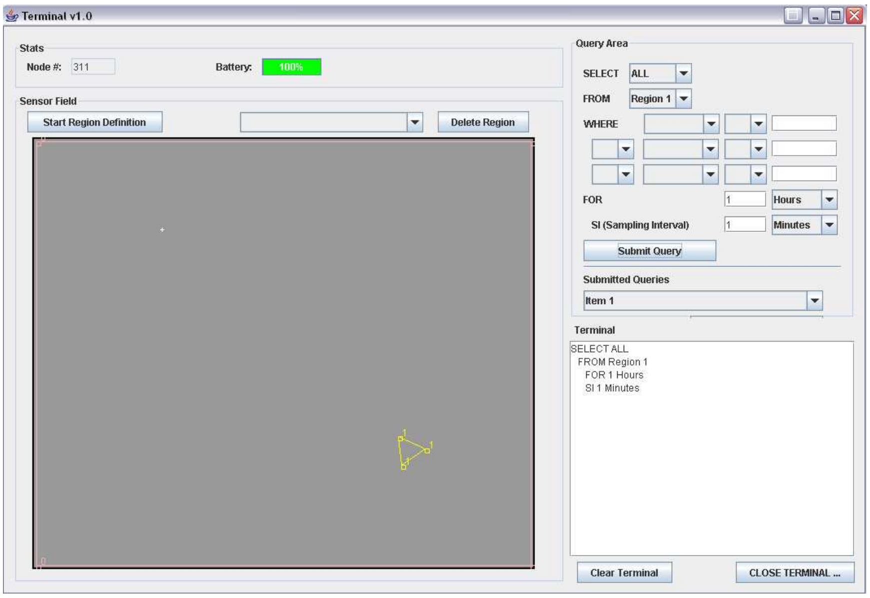

Figure 1: Terminal Window

the node's ID (integer representation of its IP) to which you have connected along with its battery status. We use infinite battery energy reserves for this example. On the left-side there is a region-drawing specification, allowing the user to create a region of interest from which samples will be acquired. The small cross existent gives you a visual cue of the relative position of "this" node relative to the deployment area. A region may be defined with a single point, at a minimum, which means that only the node closest to the region will respond to the query. Go ahead and draw a region. Make sure to click "End Region" at the end of your drawing. The regions are automatically indexed and can be referred through the SQL-like builder in the right-hand side of the terminal. Region 0 is already built and designates the entire network. Once the region definition is completed, specify the following query

SELECT ALL
FROM REGION 1 HOURS
SI 1 MINUTE

and click "Submit Query". Don't close the terminal, as you may want to visualize the result of the query as it unfolds. Look at the simulation GUI to see it developing. You may want to speed it up a little bit from the bottom-right speed-control module. The samples will arrive every 1 minute. If you want to visualize the phenomenon that is being sampled, right click on an empty region in the SIDnet GUI, and click "Show/Hide Phenomena Layer".

Once you have familiarized yourself with the run-time interaction, you may want to look at the driver, application and routing files to see how the simulation is put together. You may use these files as ”templates” to develop other applications. These files are commented so you should be able to roughly get a better idea how the SIDnet and application development goes from within.

4 SIDnet Architecture

4.1 Code Structure

SIDnet's package is placed under $SIDNETHOME$/src directory, and the source code directory structure is organized in the following way:

```
../sidnet/stack/app      - implement here the code logically
                           corresponding to the application layer of
                           the ISO network stack
../sidnet/stack/routing  - network/routing algorithms
../sidnet/stack/mac      - MAC protocol implementations
../sidnet/stack/driver   - entry-point to a simulation, where the
                           user constructs the network stack by
                           indicating the corresponding algorithmic
                           implementations, plug-in tools and
                           utility-views to be used at run time,
                           along with node-specific parameters,
                           such as energy consumption model,
                           battery model, phenomena model, etc.
../sidnet/models/deployment - contains deployment models
../sidnet/models/energy - energy consumption and battery models
../sidnet/senseable/    - sensing phenomenon and moving objects
                          models
../sidnet/utilityviews  - user-defined utility views (v1.0
                          distribution includes an energy-map and
                          statistical collector view)
../sidnet/core          - the SIDnet program

../sidnet/core/gui      - contains core-elements of SIDnet GUI
                          experience
../sidnet/core/terminal - contains the code associated to the
                          SIDnet's run-time "terminal"

../sidnet/core/misc     - contain core, non-graphical elements of
                          SIDnet simulator
../sidnet/core/interfaces - contain core, architectural elements
                            of SIDnet
../sidnet/core/simcontrol - contain the core simulation manager
../sidnet/batch         - contains the batching mechanism, which
                          can be used when complex and extended
                          simulations are to be performed in an
                          automatic fashion.
../sidnet/colorprofiles - contains user-defined color-profiles
                          for the SIDnet nodes colorings
../sidnet/messages      - placeholder for message/packet formats
```

The physical/radio layers are implemented in $SIDNETHOME$/importedpackages/jist-swans-1.0.6/ as part of the swans distribution

The main GUI window, which is illustrated in Figure 2, consists in the following elements:

- Sensor Field
- 2 x Utility Views
- Simulation Control interface
- Progress bar

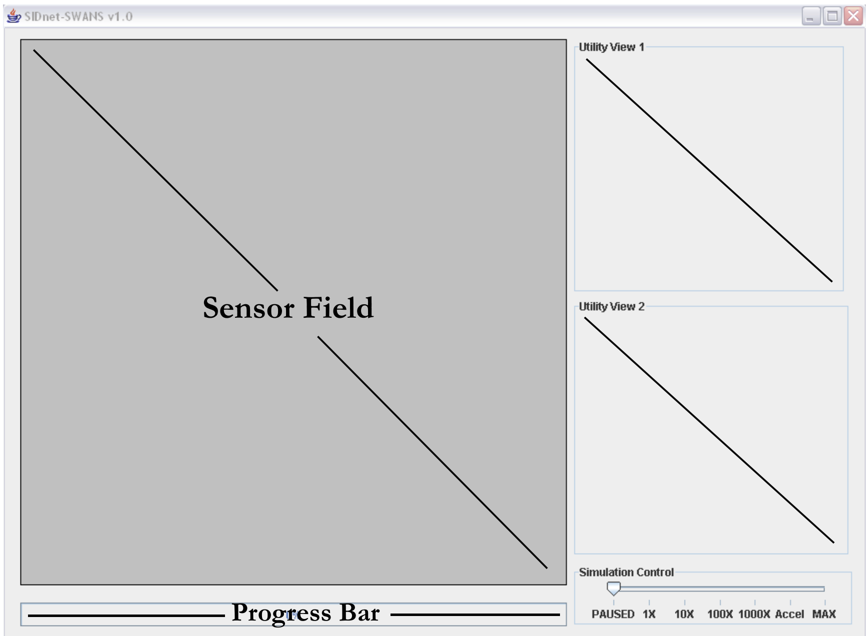

Figure 2: Main SIDnet GUI window

These components will be discusses in the following sections. A run-time sample with 500 nodes, energy map and statistical information is illustrated in Figure 3

4.2 Sensor Field GUI

The *Sensor Field* represents the container where the nodes, through their associated GUI, will be placed. Additional GUIs and plug-in tools can be integrated through the sensor field GUI, such as, for example, the group selection tool,

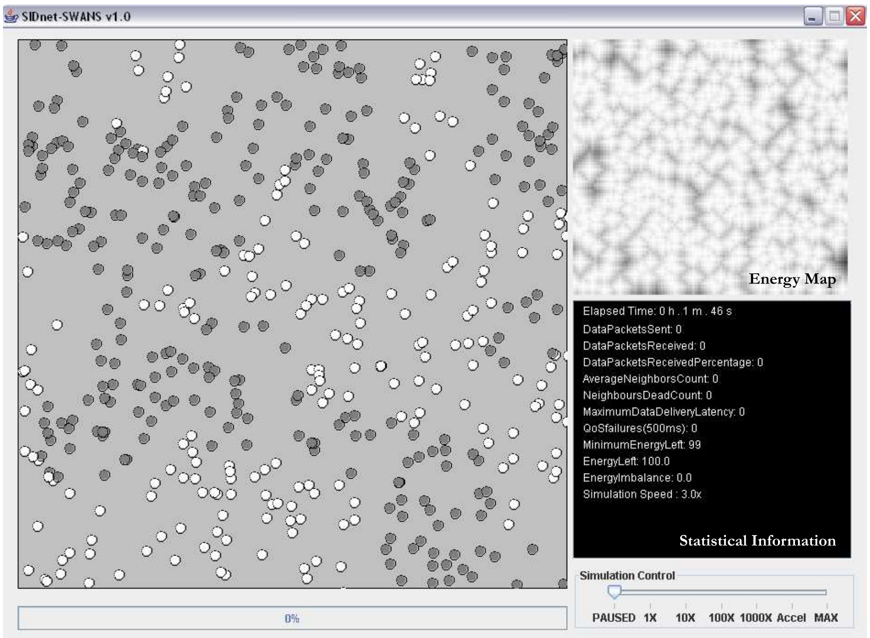

Figure 3: Sample SIDnet GUI

which allows you to select a group of sensors and perform an action over them, or phenomena models, topology visualization tools, etc, which will be discussed later on in the manual.

A sensor node is represented as a small circular object of various colors. A user can interact with the sensor node through the menu (mouse)-actions. The Sensor Field handles users' mouse interactions and forwards their actions to the appropriate listeners. If a mouse (right) click takes place over a sensor node symbol, the user will interact only that particular node through its internal menu system. If a mouse (right) click takes place outside of a sensor node symbol, the registered plug-in tools will respond to it through their own menu system. Go ahead an try to right click on the mouse symbol and then outside the mouse symbol.

4.2.1 Utility Views

The *Utility Views* are placeholders for various (user-defined) tools, such as energy map display, statistical information, etc.

4.2.2 Simulation Control Interface

The *Simulation Control Interface* allows the user to control the speed of the simulator. It is part of the core simulation manager and cannot be modified.

4.2.3 Progress Bar

The *Progress Bar* represents a convenience function which allow the user to set visual feedback regarding the progress of a particular, probably intensive, operation.

4.3 Internal Operational Architecture

Figure 4 illustrates the connection between the network stack and the GUI-side, along with other components of the SIDnet.

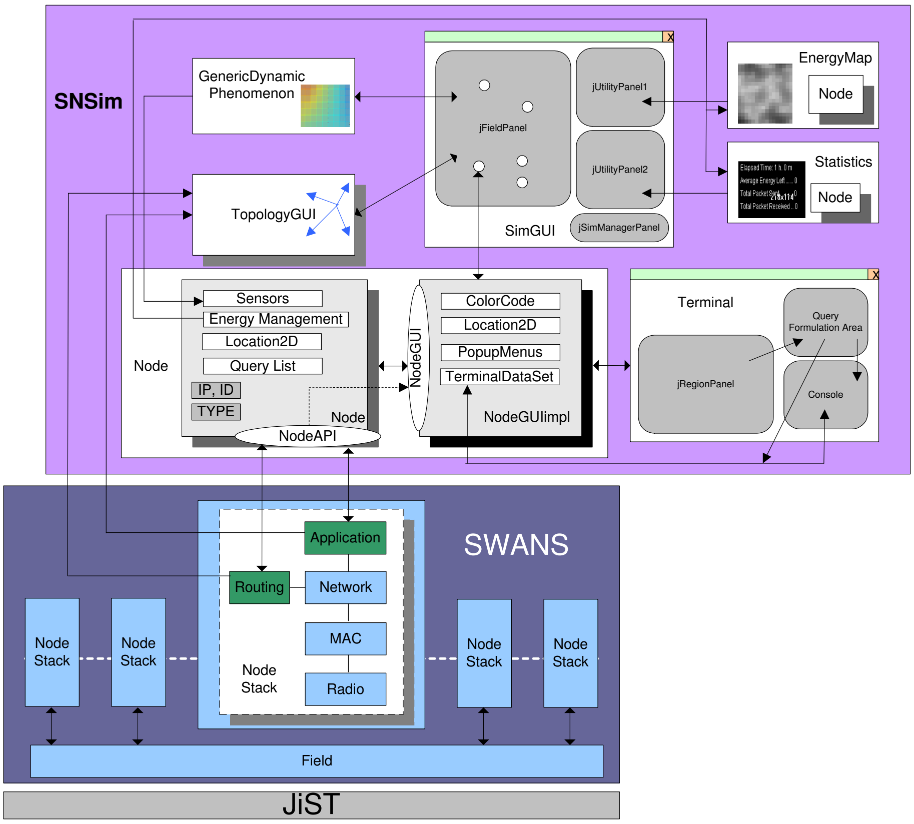

Figure 4: Operational Architecture of SIDnet

The central abstraction in the SIDnet-SWANS simulator is the "NODE", as we will refer to as the "SIDnet Node". The SIDnet node represents the interface between the network stack and all the other components of the simulator, including GUI, sensorial field, location services, energy management, etc. Each application and network/routing implementation must keep a reference to its corresponding SIDnet node. Node also represents a placeholder for information that is to be shared amongst network stack element, such as neighboring nodes list, energy levels, etc.

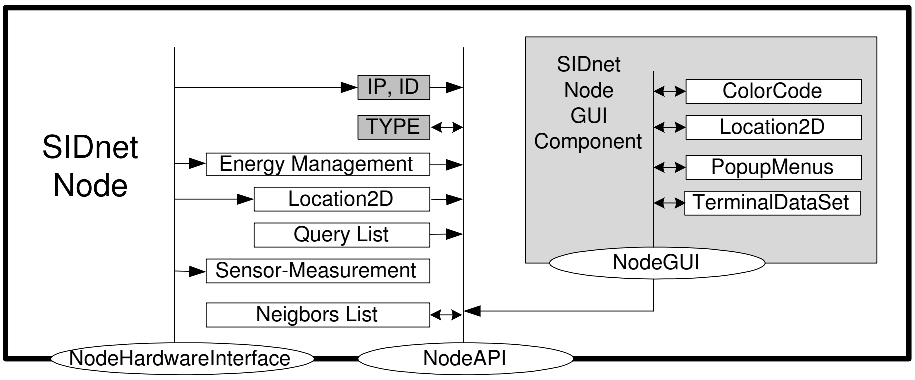

Figure 5: The API-structure of the SIDnet Node

The overall -application programmer's interface- structure of the SIDnet node is given in Figure 5. The SIDnet node has two main components:

- Network stack related component
- GUI-related component

The network stack level component contains information such as:

- IP of a node (IPv4)
- ID of a node (the numerical - integer - equivalent of the IP)
- TYPE of a node (type gives the means of identifying the category-type of a node in a heterogeneous network)
- Location (x, y) of the node in the field of sensor nodes
- Access to sensor boards for sensor readings
- Access to energy management for battery-level readings

The access to the network stack level components is done by means of two interfaces:

- NodeAPI
- NodeHardwareInterface

The *NodeAPI* interface allows the users to access, but not modify (except the TYPE and neighbors list) the data stored on behalf of the network stack level component. Each application and network layer implementations must hold a reference to a *NodeAPI* node.

The *NodeHardwareInterface* allows the users to *configure* the node, i.e. setting the IP address, the sensor boards, changing/refilling the battery, changing the location, etc. The contents of the Node should be accessed for modification through the NodeHardwareInterface only through the Driver file.

The GUI-related component contains information related to GUI only. Such information includes:

- Location (x, y) of the node on the SCREEN !!! (not in the field), expressed in pixels
- ColorCode - controls the "color" of the node as it is displayed on the screen
- Menus
- Terminal interface

Please refer to the JAVADOC references on the webpage. Specifically, look for:

- sidnet.core.interfaces.NodeAPI
- sidnet.core.interfaces.NodeHardwareInterface

The related implementing classes are the following:

- sidnet.core.misc.Node
- sidnet.core.gui.NodeGUIImpl

5 Navigating through SIDnet-SWANS's network stack

The SIDnet-SWANSs network stack, which is illustrated in Figure 6, is a subset of the typical network stack found in wired networks and also in SWANS. Given the specifics of the wireless sensor network however, the Transport Layer, which was present in the original JiST-SWANS distribution, was not inherited in SIDnet-SWANS.

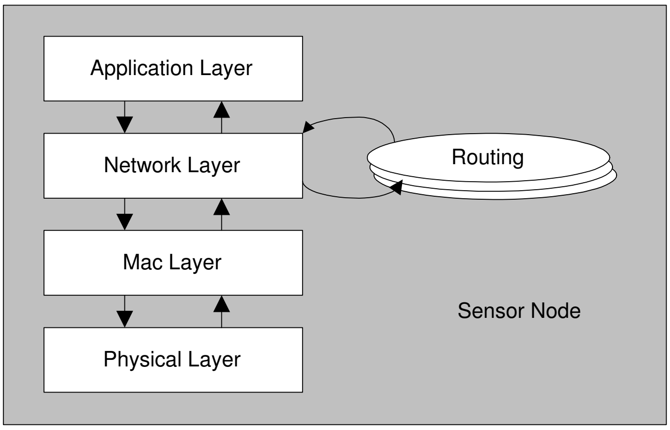

Figure 6: SIDnet-SWANS's network stack

The Network Layer represents a switchboard between packets coming from the upper layers (Application Layer), lower layers (Mac Layers) and the Routing paradigms, and based on the destination address of the messages, it forwards the packets accordingly. Figure 7 illustrates the overall message flow between the Application Layer, Mac Layer, Network Layer and Routing.

As it can be observed, most of the messages are flowing through the Routing element. Imagine that a users application may need to send a data-packet to a distant node, located couple of hops away. The application may know the IP address of the destination node, but not the means of reaching it (the route). The Network Layer will let the Routing compute and retrieve the address of one (or more) of the neighboring (1-hop) nodes to which the message to be sent immediately in its route to the destination. The single exception to this rule applies to outgoing Broadcast messages, which are detected by the Network Layer and forwarded directly to the Mac Layer (there is no need for routing since broadcasting means transmitting to whoever can hear the message within communication range). But since routes may be built considering all types of messages, ALL the incoming messages however (both unicast and broadcast ones) are sent directly to the Routing algorithm, where they may be used either for updating the routing information, finding the next-hop node if the packet has not reached its final destination, or passed up, to the Application Layer, if

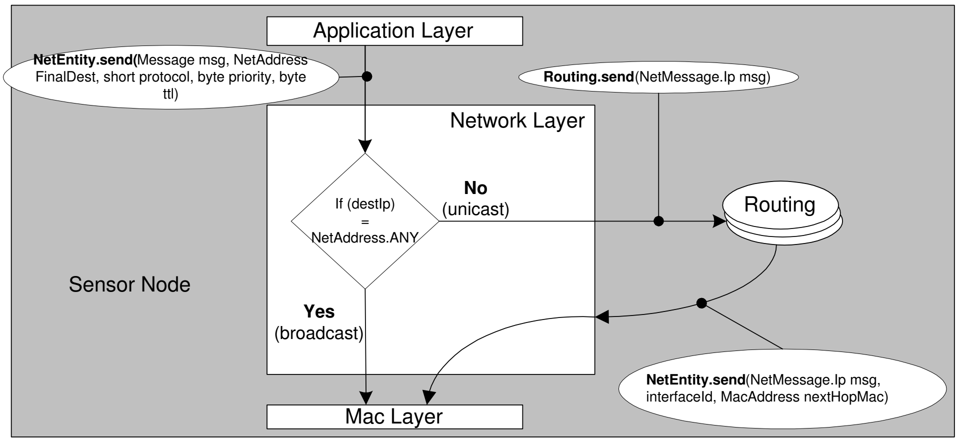

Figure 7: Overview of the message flow in the upper layers of the network stack

and only if the message destination is the node itself.

Figure 8 gives a detailed description of the outgoing flow of messages and the associated methods that are being called within the corresponding .java files.

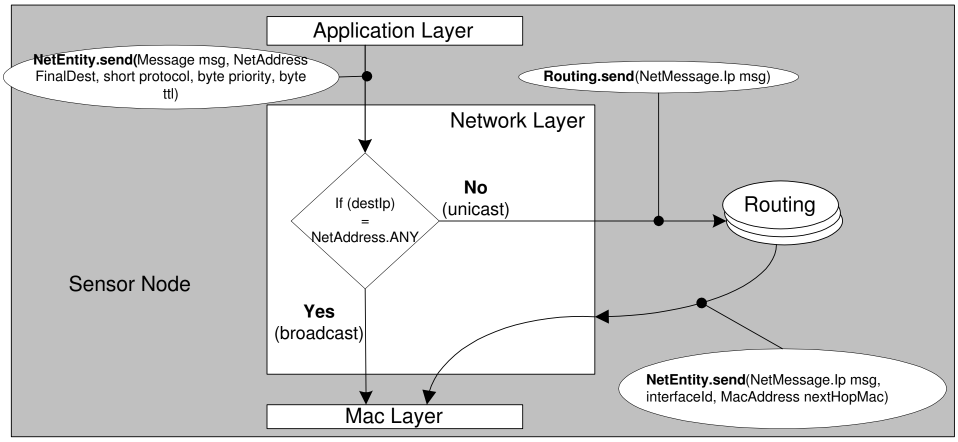

Figure 8: **Outgoing** message flow in the upper network stack

In order to send a message from the Application Layer, the following parameters are needed:

- `Message`: a user defined pojo containing the data to be transmitted (the payload)
- `NetAddress`: the IP of the node that represents the FINAL destination of this packet

- `Protocol`: if there are several routing protocols implemented, this is used to specify the routing protocol that is to be used for routing the packet
- `Priority`: indicate if some packets have a higher priority than others
- `TTL`: time to leave: in a congested network, indicate the amount of time a packet will be held before being dropped if continuous attempts of forwarding the data fail.

The application layer will call `NetEntity.send( )` with the above parameters specified. For example, broadcasting a message can be specified as:

```
NetEntity.send(myMessage, NetAddress.ANY, Constants.MY_PROTOCOL, 1,
(byte)100)
```

Or, unicasting to a known IP (net) address:

```
NetEntity.send(myMessage, destinationNetAddress, ...)
```

If the outgoing message represents a broadcast, it will be sent directly to the MAC layer, bypassing any routing primitives. However, if it is a unicast message, it will be passed to the Routing algorithm to be handled. In the latter case, the NetEntity will call the following method member of the Routing class:

```
Routing.send(NetMessage.Ip ipMsg)
```

The user must decide what to do with the message at the routing layer. Most likely, it needs to find the 1-hop neighbors MAC address to forward the packet toward its final destination. Note that the NetEntity will ship a wrapped version of the original message the application layer has sent. To obtain the content, use the .getPayload() method

```
MyMessage myMessage = ipMsg.getPayload()
```

Once the MAC of the next-hop neighbor has been retrieved, the routing layer can proceed sending the packet down the stack to the MAC layer. However, since it does not have access directly to the MAC Layer, it must rely on the following method of the Network Layer

```
netEntity.send(ipMsg, interfaceId, MacAddress)
```

Do note that the Network Layer will require the wrapped version of the payload, not the payload itself (that is, a NetMessage.Ip formatted message, which will contain also information about the original source of the message and its final destination). The user needs to use for the interfaceId parameter the default interface, which is indicated through `Constants.NET_INTERFACE_DEFAULT`. The `MacAddress` corresponds to the next-hop node the packet will be forwarded to. If the IP address of the neighbor to forward the data-packet is known , you may use the `neighboursList` (`NodesList`) to retrieve the associated Mac address as follows:

```
neighboursList.get(nextHopDestIP).mac
```

where the `neighboursList` is automatically preloaded by the hearbeat protocol that is executed in the first hour of simulation time.
WARNING! The following method may be also called from the Routing Layer

```
NetEntity.send(myMessage, NetAddress.ANY, Constants.MY_PROTOCOL, 1,
(byte)100)
```

Since it is a broadcasting message, it will be send immediately to the Mac layer. However, if it wouldn't be a broadcasting message, the Network Layer will NOT send the packet to the MAC Layer. Instead, since the Network Layer is unaware of the origins of the call, it will handle the message back to the Routing Layer, just as if the Application Layer has called it, risking creating an infinite message-passing loop between the Routing and Network Layer. Figure 9 represents a detailed illustration of the incoming flow of messages and the associated methods that are being called in the corresponding `.java` files.

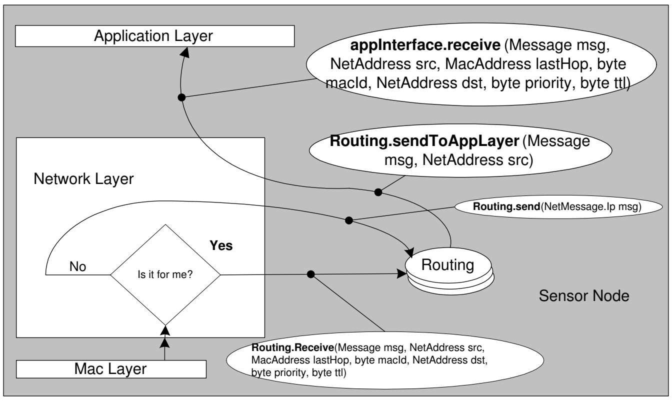

Figure 9: **Incoming** message flow in the upper network stack

The Network Layer, upon receiving a message from the Mac Layer, will check to see if the hosting node represents the final destination of the packet. If it is not, meaning that the current node is just a relay of the packet in its way to the final destination, the Network Layer will ask the routing protocol to send the packet again (aka, forward). If this node represents the final destination of the packet, it will be treated through the receive method of the Routing Layer, in which the packet must be treated, and, in most cases, sent to the application layer as well. Note that, the Network Layer will not forward any packet to the application layer by itself. Here is the receive method syntax the Network Layer will call the following method of the Routing class:

```
Routing.Receive(Message msg, NetAddress src, MacAddress lastHop,
byte macId, NetAddress dst, byte priority, byte ttl)
```

The send() method is the same one called when unicasting a packet from the Application Layer. Indicating the content of the message, the IP of the original

producer of the message, the last hop Mac address the message is coming from, the Mac interface it has been received through and the final destination of the packet. These may be used by the Routing layer to decide if the packet has reached its final destination. The routing layer will decide whether the incoming data packet is to be forwarded again, and/or update its routing information if necessary. If the hosting node does not represent the final destination (represents just an intermediate node on the route) of the data packet, the Routing Layer must forward the message to the next node on the route following the indication for outgoing messages. Otherwise, it should pass the message to the Application Layer, which can be done directly by calling the following locally defined method:
`Routing.sendToAppLayer(Message msg, NetAddress src)`

In turn, the Application Layer will be notified by the incoming message by being called its local method, which follows:
`appInterface.receive(Message msg, NetAddress src, MacAddress lastHop, byte macId, NetAddress dst, byte priority, byte ttl)`

6 SIDnet Operations and Tools

6.1 SIDnet Node - coloring

A SIDnet node is represented in the GUI through a circular icon. You cannot control the size of this icon, but you can control its *coloring-scheme*. Namely, you can control the colors of the *contour line* and *body* of the icon, as illustrated in Figure 10:

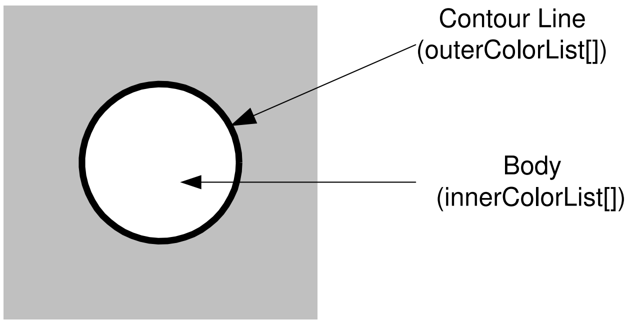

Figure 10: (Color)-Controllable elements of the SIDnet node icon

User can define a set of possible colors and their meanings through a color-profile that you can associate with your application. The color-profile can be assigned in the Driver file.
The user-defined color-profiles must be placed under

$sidnet.colorprofile$

package. A "generic" color code profile can be found under

$sidnet/colorprofile/ColorProfileGeneric.java$.

User-define color profiles can be created programmatically, by writing a .java class that extends the following *abstract* class:
$sidnet.core.interfaces.ColorProfile$
A color-profile contains a set (array) of color-schemes. A minimum set of colors are already hardcoded in the ColorProfile.java classs. At a minimum, a color profile should associate colors for the following operations:

- color of the node that is dead (default: black/black (line/body))
- color of the node that is alive and listening (default: black/white)
- transmitting node (red/red)
- receiving node (green/null)

- ...

A user can overwrite these colors and define its own coloring schemes.

The (possible) colors are stored as two (java.awt.Color) arrays, as illustrated in Figure 10 (see also the sidnet.colorprofiles.ColorProfileGeneric.java):

- Color[] innerColorList - corresponds to the *body* color
- Color[] outerColorList - corresponds to the *contour line* color of the node icon

The index in these lists, which can be statically defined (i.e., public static final int DEAD = 0;), designate the meaning of a particular color-scheme. Clearly, this means that the contour/body color combination for the node being dead can be retrieved from innerColorList[DEAD] and outerColorList[DEAD] respectively.

The rank of the colors in the arrays define the "priority" of the colors-schemes. Lower indexes have higher priorities. Higher priorities "overwrite" (or mask) lower priority color-schemes. This is useful when you apply two color-schemes to the node in the same time and decide which one of the two will be seen on the screen. Obviously, you cannot have two colors of the contour line applied in the same time. For example, let's say you have a a sink-node, represented as black/red combinations. If your sink-node is receiving a message, you may want to see that node temporarily being colored as black/green to visually indicate the receival of a message. This can happen if you place the black/green combination at an index that is lower than the index of the sink-node color-scheme in the array. Otherwise, the black/green will not be seen on the screen. For example, consider the following code-association:

innerColorList[RECEIVE] = Color.GREEN;
outerColorList[RECEIVE] = Color.BLACK;

innerColorList[SINK] = Color.RED;
outerColorList[SINK] = Color.BLACK;

If the RECEIVE is 1 and SINK is 2 ($RECEIVE < SINK$), then you will see when the sink node receives a message. If the RECEIVE is 2 and SINK is 1 ($RECEIVE > SINK$) you will not see when the sink node receives a message.

You must always have a DEFAULT color-scheme, which should have the lowest priority. The color-schemes indexes must be defined in consecutive numeric order, from 0 to $(DEFAULT - 1)$

6.1.1 NULL Colors

You may also define "NULL" colors. Null colors act as "transparent" colors. If, for example, you don't care what should be the color of the contour line when performing an action, specify it as null.

6.1.2 Temporal-validity of a color-scheme

You can specify the period of time a particular color-scheme applies to the node. For example, you may indicate by a brief "blink" of a color when a node receives a message. The coloring-time is entirely controlled by the SIDnet simulator. It is up to you, however, to specify the "amount" of time. The following are the time-markers associated to a color-scheme:

- ALWAYS - apply the color-scheme from now-on for an undefined amount of time
- CLEAR - cancels the effect of the "ALWAYS"
- ms-value : the interval of time, expressed in milliseconds for which the corresponding color-scheme is valid. The SIDnet will deactivate the color-scheme automatically when the interval of time expires

The temporal-markers are specified in the

sidnet.core.interfaces.ColorCode.java interface.

6.1.3 What is the simplest way to define my own color profile?

First, create a copy of the ColorProfileGeneric.java and rename as you wish. Then modify its content and define the colors that you need for your application. You can increase the number of tags and follow the instructions included in the ColorProfileGeneric.java file.

You can permanently assign a color profile to a node in the Driver file through

node.setColorProfile() function.

and retrieve it later through

node.getColorProfile() function

Or, you can just access it directly. For example, to color a node, you can use the following code:

node.getNodeGUI().colorCode.mark(ColorProfileUser.DEAD, ColorProfileUser.FOREVER).

6.1.4 Run-time usage

Once you have properly defined a color-profile, you can access these profiles as follows:

node.getNodeGUI().getColorProfile().mark(SINK, ALWAYS)
node.getNodeGUI().getColorProfile().mark(RECEIVING, 500 /*ms*/)

node.getNodeGUI().getColorProfile().mark(DEAD,ColorProfile.ALWAYS)

See the $sidnet.core.interfaces.ColorProfile$ API on the website for the outline of the possible functions.

7 SIDnet Run Modes

SIDnet can run in three modes:

- DEBUG
- DEMO
- EXPERIMENTS

By convention:

- "DEBUG" mode should be used to display both GUI and command-line debugging information
- "DEMO" mode should be used to "hide" most of the unnecessary, cluttered debugging information, retaining only what is necessary to do a "slow-speed" demonstration
- "EXPERIMENTS" mode should be used for high-speed experimental evaluation. Under this mode, you should programmatically set the simulator speed to MAX (disables graphics)

These modes are only conventions and must be implemented programmatically. They can be set as system variables in Java.

8 Collecting Run-Time information: The "StatsCollector" utility view

Package: *sidnet.utilityvies.statscollector*

What is it?

The "StatsCollector" is an utility view (aka plug-in) for SIDnet which allows you to visualize and collect (log) run-time information regarding the experiment that is currently being performed. It typically shows on the lower-right corner of the SIDnet's GUI.

What type of information can be collected?

The StatsCollector can collect the following information:

- Time (time-stamps)
- Number of data packets sent
- Number of data packets received (at their destination)
- Percentage of data packets that have been received (considering the ones that have been sent),
- Number and percentage of data packets that have been lost/dropped
- Packet delivery latency information (average, minimum, maximum)
- Number and percentage of nodes alive/dead
- Energy left (average, minimum, maximum, stdev) in relation to the entire network
- Number of 1-hop neighbors discovered (average, minimum, maximum) both network-wide or within a specified region.

How do I use/configure it?

You must define it in the Driver file of your experiment, within the the `public static Field createSim(...)` function.

It is a three step process:

1. Instantiate the StatsCollector;
2. Configure it
3. Register it with SIDnet interface

8.1 StatsCollector instantiation

You need to provide the following information to *StatsCollector*'s constructor:

- **Node[]** - the array of nodes. StatsCollector needs this to query the status of each of the nodes. Remember, each sensor node (and its stack) have a *Node* associated with it.
- **battery capacity** (assuming you are running a energy-aware application; otherwise set this to 0);
- **field length [fts]** - the length of the (assumed-squared) field in which you deploy the nodes. This parameter it is usually supplied as a command line argument to the driver.
- **sampling interval** - the interval of time you wish data to be logged. It is expressed in SIDnet atomic time. A typical value is 30 minutes (30 * Constants.MINUTE)

If you plan to collect statistical information by means of SIDnets batching mechanism (see Section 9), you may provide, in addition, the following information so that it will be easier to organize log-files from various experiments and/or runs.

- **runId** - if you perform batching, runId will be provided as a command line argument. Use it here. If not, set this to 0;
- **experimentId** - follow the same instructions as for "runId"
- **experimentsTargetDirectory** - the directory where the log-files will be created and stored. Use "." for the default, currently working directory (most likely /SIDnet-SWANS).
- **Prefix** - use this to specify a prefix to the log file names. You may set this to "" if not interested to add a prefix.

Example (non-batching):

Assume that `Node[] nodesList` was created. Then,

statistics = new StatsCollector(nodelist, (int)battery.getCapacity(), length, 30 * Constant.MINUTE);

Example (with batching)

statistics = new StatsCollector(nodelist, (int)battery.getCapacity(), length, 30 * Constants.MINUTE, 1 /* runId */, 1 /* experimentId */, "C:
experiments" /* experimentsTargetDirectory */, "newRoutingExperiment-");

8.2 StatsCollector Configuration

`StatsCollector` utility view needs to know what exactly you are interested to monitor/log. You can do so through the `.monitor(...)` member function. Due to run-time efficiency reasons, some of the functionality is provided within `StatsCollector` class (such as packet monitoring), which can be referred to by means of "ITEM"s, other is "externalized" as "StatEntry" classes (such as nodes/neighboring/energy monitoring).

Here are the defined ITEMs that you can use:

```
public static enum ITEM
{
TIME,
DATA_PACKETS_SENT_COUNT,
DATA_PACKETS_RECEIVED_COUNT,
DATA_PACKETS_RECEIVED_PERCENTAGE,
DATA_PACKETS_RECEIVED_COUNT_CONDITIONAL,
DATA_PACKETS_RECEIVED_PERCENTAGE_CONDITIONAL,
DATA_PACKETS_LOST_COUNT,
DATA_PACKETS_LOST_PERCENTAGE,
DATA_PACKETS_AVERAGE_DELIVERY_LATENCY,
DATA_PACKETS_MAXIMUM_DELIVERY_LATENCY,
DATA_PACKETS_MINIMUM_DELIVERY_LATENCY,
}
```

Here is also the list of *StatEntry* classes (as in SIDnet v.1.2.0):

- `StatEntry_AliveNodesCount`
- `StatEntry_AverageEnergyLeftPercentage`
- `StatEntry_AverageNeighborsCount`
- `StatEntry_MinimumNeighborsCount`
- `StatEntry_DeadNodesCount`
- `StatEntry_DeadNodesPercentage`
- `StatEntry_EnergySTDEV`
- `StatEntry_MaximumEnergyLeftPercentage`
- `StatEntry_MaximumNeighborsCount`
- `StatEntry_MinimumEnergyLeftPercentage`
- `StatEntry_MinimumEnergyLeftPercentageWithinRegion`

More `StatEntry` may be added, so you may want to check `SIDnet-SWANS/src/sidnet/utilityviews/statscollector` folder. Verify the API for these `StatEntry`-es for usage.

Note: The order in which you add to `.monitor(...)` is the order in which it will appear both on the screen and also in the log-files.

Example:

```
statistics.monitor("Time", StatsCollector.ITEM.TIME);
statistics.monitor("DataPacketsSent",StatsCollector.ITEM.DATA_PACKETS_SENT_COUNT);
statistics.monitor("DataPacketsReceived",StatsCollector.ITEM.DATA_PACKETS_RECEIVED_COUNT);
statistics.monitor("DataPacketsReceivedPercentage",StatsCollector.ITEM.DATA_PACKETS_RECEIVED_PERCENTAGE);
statistics.monitor(new StatEntry_AverageNeighborsCount());
statistics.monitor(new StatEntry_DeadNodesCount(1));
statistics.monitor("MaximumDataDeliveryLatency",StatsCollector.ITEM.DATA_PACKETS_MAXIMUM_DELIVERY_LATENCY);
statistics.monitor(new StatEntry_MinimumEnergyLeftPercentage());
statistics.monitor(new StatEntry_AverageEnergyLeftPercentage());
statistics.monitor(new StatEntry_EnergySTDEV());
```

Note: you can provide header/tag information to identify data (it will correspond to heading in excel) for `StatEntry`-es as well. While it uses a default header title, you can change it. For example, you can specify:

```
statistics.monitor(new StatEntry_DeadNodesCount("NumberOfDeadNodes",1));
```

instead of

```
statistics.monitor(new StatEntry_DeadNodesCount(1));
```

for which the default heading/tag is `"DeadNodesCount"`.

8.3 Register StatCollector with SIDnet

This last step that **MUST** be done for the `StatCollector` to receive timing-calls and record data appropriately. It also specified in which (of the two) utility-views windows in the SIDnet GUI to appear.

If you want the statistics to appear on the lower-right window of SIDnet GUI (recommended), use this:

```
simManager.register(statistics, simGUI.getUtilityPanelContext2());
```

or, for the upper right cornet, use this:

```
simManager.register(statistics, simGUI.getUtilityPanelContext1());
```

9 Batching

Nomenclature:

- SIDnet experiment: The execution of a single SIDnet instance, which is parameterized (and thus configured) through the driver file
- SIDnet run : The execution of a set of SIDnet distinct experiments. By distinct experiments we mean either
  - having different driver files, or
  - having the same driver file whose parameters are changed.

How does it help me?

The SIDnet Batching mechanism allows you to perform quickly (over-night), without your explicit intervention, a large number of SIDnet experiments (hundreds, maybe thousands).

Where are the Batching-related files?

Under the `SIDnet-SWANS/src/sidnet/batch` directory (`sidnet.batch` package)

How does it work?

The Batching mechanism configures each SIDnet experiment by feeding command line arguments to the driver file for the current SIDnet experiment which is going to launch. Upon termination of a SIDnet experiment, the Batching mechanism will launch a new SIDnet experiment with a new set of parameters.

It works in conjunction with the "Statistics" utility views (see Section 8) which is in charge of storing run-time information to log-files.

OK, how do I do all these?

Follow these steps:

- Configure Environment
- Build the parameters file
- Configure the Driver file
- Launch the Batching Mechanism

9.1 Configure Environment

1. Make sure `java.exe` is in the PATH [1]

2. Create an empty folder anywhere you wish (for example, "`C:/experiments`"). This is were all the results and run-time log data will be stored and organized

3. Copy "`setenv.cmd`" and "`checkenv.cmd`" from

4. Edit "`setenv.cmd`". Namely, change the "`SIDNETSWANSDIR"=C:/your installation path/SIDnet-SWANS`"

5. Open a command-line window

6. Navigate to your /experiments directory

7. Set your environment (launch "`setenv.cmd`" from the command-line)

8. Leave the command-line window opened.

9.2 Build the parameters file

The easiest (and recommended) way is to use Excel (or similar tabular spreadsheet software).

Conventions:

Columns: hold various parameters of a given experiment

Rows : each row contains the parameters of a SIDnet experiment

First ROW is a header row and should contain the name of each of the parameters underneath. The Batching mechanism will start picking up run-time parameters and executing SIDnet experiments starting with the second row.

First three (3) columns MUST contain the following information

- Col #0: "Driver Filename"
- Col #1: "expId" (experiment Id).
- Col #2: "runMode"

[1] See Q & A on how to set this if you don't know how

Table 9.2 contains a sample (template) for writing the parameter file

Driver Filename	**expId**	**runMode**	Nodes	Area[ft]	timeout	other params ...
$sidnet.stack.driver.Driver_{s}ampleP2P$	1	experiments	500	5000	1000000	...
$sidnet.stack.driver.Driver_{s}ampleP2P$	2	experiments	400	5000	1000000	...
$sidnet.stack.driver.Driver_{s}ampleP2P$	3	experiments	1000	7000	1000000	...
$sidnet.stack.driver.Driver_{s}ampleP2P$	4	experiments	700	6000	1000000	...
...	...	...	...	...	1000000	...

When done, export the file to .csv format and save it under the `/experiments` directory.

9.3 Configure the Driver file

The Batching mechanism will parse the .csv file, line by line, and supply the arguments (starting with the second column) to your Driver file. YOU MUST modify your driver file to configure local variables according to the supplied command line arguments.

9.4 Launch the Batching Mechanism

In your command-line window, which now has the environment set-up, assuming you are still under the `/experiments` directory, use the following command to launch the batcher

```
>java sidnet.batch.SIDnetCSVRunner <fileName>.csv [-runid=#] [-experimentid=#]
[ -demo | -experiment ] -parallelism=#
```

where

- `<filename.csv>` is the .csv filename you have created
- `-runid=#` - OPTIONAL, specify an integer number that will help you identify this run. If you do not specify one, the batching mechanism will generate one for you. The experimental results associated to a given run will be stored under /experiments/run#, which will be automatically created for you.
- `-experimentid=#` - OPTIONAL, use this only if you want to run ONE experiment out of the many specified in the .csv file. Each SIDnet experiment will have a unique file created for it under the `/experiments/run#`
- `-demo | -experiment` - refer to Section 7
- `-parallelism=#` - indicate the number of SIDnet experiments that you allow to run in parallel. This is a good feature to use when having a multi-core machine. The rule of thumb is to specify the number of cores you processor has.

examples:

```
java sidnet.batch.SIDnetCSVRunner myExperiments.csv -parallelism=1

java sidnet.batch.SIDnetCSVRunner myModifiedExperiments -runid=2 -parallelism=2
```

A GUI window will pop-up. Verify that the information is correct, then click START. Sit back and relax.

9.5 Interrupting a SIDnet Batched Run

The SIDnet batching mechanism has been designed that it can be interrupted whenever you need. All the finished experiments will be saved. All the information (logs) corresponding to the experiments that were "still" running (unfinished) will be deleted. You may resume the experiments by re-launching the Batching mechanism (but make sure you specify the same `run#` you specified - or has been assigned - to the run that was previously interrupted). The batching mechanism will not replace already saved log-files and will proceed from where it was previously interrupted. If you specify a different `run#`, a new directory will be created under the /experiments directory and experiments will start from the beginning. This is useful when you want to repeat the same experiments a couple of times and "average" the results.

10 Q & A

1. Q: What java version do I need? A: 1.5 or later

2. Q: How do I know if (or the correct version of) ”java.exe” is in the PATH? A: Launch a command-line terminal (WindowsXP: Click START-¿Run-¿Type ”cmd”-¿Hit Enter). Type ”java -version”-¿Hit Enter at the command prompt. If everything is configured correctly, you should see the version of the java that you are currently using.

3. Q: I have installed the correct java jdk version, and I know location where it was installed. How do I add it to the path? A: You have to perform this operation only once. Under WindowsXP: START-¿Settings-¿Control Panel-¿System-¿Advanced-¿Environment Variables-¿Under ”System Variables” locate the ”Path” entry (or create one if not present)-¿Edit. Add at the end of it ”;” followed by the full path to the ”java.exe” file. Hit OK and close all the windows.

11 Troubleshooting

- **Problem**: At run-time, I get a "class not found" error messages.
  Solution:

  1. Is the **.java* file in the build path?

  2. Check the build directory (typically "SIDnet-SWANS\ build") and try to locate manually the *.java. If you cannot locate the file, check the (**1**). If you can locate the file, go to (**3**)

  3. Is the **.class* in the CLASSPATH? Check your environment variables (*setenv.cmd*) to point correctly to the source of the sidnet package. If the CLASSPATH is correct, then goto(**4**)

  4. Is the error similar to the following:

  JiST class loader: class not found [Lsidnet.utilityviews.StatsCollector$ITEM]

  then, you might be using an old "BCEL" library. Make sure your environment is set up to use the "BCEL v5.2" which is under "SIDnet-SWANS\libs", and not the "BCEL v5.1" which is part of the JiST-SWANS distribution ("SIDnet-SWANS\importedpackages\ jist-swans-1.0.6\libs\bcel.jar"). The older version of BCEL is known to create this type of problems at run-time.

- **Problem**: I have also downloaded and installed NetBeens IDE for testing SIDnet-SWANS with it. But after building project according to your instructions manual, I have got the following errors:
  init:
  deps-jar:
  Compiling 267 source files to E:/SIDnet-SWANS/build/classes
  E:/SIDnet-SWANS/importedpackages/jist-swans-1.0.6/src/jist/runtime/JistAPI.java:92:
  package org.apache.bcel.classfile does not exist
  $org.apache.bcel.classfile.JavaClass\, process(org.apache.bcel.classfile.JavaClassjcl)$
  $throwsClassNotFoundException;$
  E:/SIDnet-SWANS/importedpackages/jist-swans-1.0.6/src/jist/runtime/JistAPI.java:92:
  package org.apache.bcel.classfile does not exist
  $org.apache.bcel.classfile.JavaClass\, process(org.apache.bcel.classfile.JavaClassjcl)$
  $throwsClassNotFoundException;$
  E:/SIDnet-SWANS/importedpackages/jist-swans-1.0.6/src/jist/runtime/Rewriter.java:12:
  package org.apache.bcel does not exist
  import org.apache.bcel.*;
  ....
  Solution
  You somehow forgot to add the following library in NetBeans, which is tied to the above errors:
  SIDnet-SWANS/libs/BCEL/org/apache/bcel-5.2/bcel-5.2.jar