HIGH LEVEL ABSTRACTIONS AND VISUALIZATION OF SENSOR NETWORK APPLICATIONS

by

SANDEEP PULLURI

B.E., Osmania University, India, 2005

A THESIS

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
Dr. Gurdip Singh

Abstract

TinyOS is a component based operating system written in nesC programming language. TinyOS provides interfaces and components for common low level abstractions such as packet communication, routing and sensing for node level sensor network application programming. This project aims to provide high level abstractions to users by providing the notion of a virtual node, which represents a set of physical nodes, allowing users to specify global scenarios, and a mechanism to decompose a high level global scenario into local node level scenarios for each of the individual sensor nodes.

A global scenario with virtual components, provided by the user, is first converted into a global scenario by eliminating the virtual components from the model by using a mapping information provided the user and replacing these virtual components by their respective physical components. Appropriate algorithm components and the automatically generated adapter components for these algorithm components are then plugged-in to implement inter-node interactions. This global scenario is then converted to the node level local scenarios by introducing the automatically generated proxy components for the remote components and connecting these proxy components using the RMI layer. The Cadena model is modified to include the attribute location for the components to identify the remote components. The make files are then generated for these local scenarios and are ready to be deployed on the physical motes.

The framework provides a GUI tool which is used to visualize the data of the sensor network in both simulation and deployment. The framework provides the user with commands that can be issued to the network from the Cadena component model as a set of interfaces to the components and a python script is used to capture this information in an xml file. The Cadena model is modified to include the attribute observable to the interfaces to identify them as the GUI commands. The GUI loads this XML file and the topology file for the actual deployment, can issue commands to the network and displays the results to the user. The GUI tool also enhances the Tossim simulator to model the external effects over the sensor network and to place the motes based on the topology information using the Tython environment.

Table of Contents

List of Figures

Acknowledgement

I would like to take this opportunity to express my gratitude to some important people who have inspired and guided me to complete this project.

Firstly, I thank my Major Professor and Advisor, Prof. Gurdip Singh for his continued support and guidance throughout this project. I thank you for your patience, for always being supportive and willing to point me in the right direction.

I thank my committee members, Prof. Daniel Andersen and Prof. Mitchell Neilsen for their support.

Finally, I would like to thank my family and friends for their unrelenting support and encouragement, and for raising my spirits whenever I needed it.

CHAPTER 1 – TinyOS

1.1 Introduction

TinyOS is an open source component based operating system designed for wireless sensor networks. It features a component-based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks. The TinyOS system, libraries, and applications are written in nesC, a dialect of C programming language optimized for the memory limitations of the sensor networks. nesC supports the TinyOS concurrency model and its programs are a set of software components which are connected to each other using interfaces.

1.2 Component Model

A nesC application consists of components which can use or provide interfaces and different components are connected using these interfaces. An interface in a nesC application consists of commands and events. A component which provides the interface has to provide the implementation for the commands in that interface and can signal the events to the components using that interface. On the other hand, a component using the interface has to provide the implementation for the event handlers in the interface. Modules in TinyOS provide the implementation of the components and the configuration. A scenario is a collection of components and the wiring between the interfaces of these components which describes the complete application.

NesC programs are built out of software components some of which are hardware abstractions. TinyOS provides interfaces and components for common abstractions such as packet communication, routing and sensing. The framework developed in this project provides the user with higher level abstractions with some generic services so that they can be directly used. It also aims at providing the user a global view of application while abstracting the underlying communication details.

CHAPTER 2 – Tossim

2.1 Introduction

Sensor networks are composed of a large number of tiny communicating devices (motes) with the capability of sensing and computation. Compared to traditional networks, the motes in sensor networks have very limited computational and communication capabilities because of their low energy resources.

Tossim is a discrete event simulator for TinyOS sensor networks. The TinyOS code for a mote can be directly compiled to the TOSSIM framework which runs on PC. In this way, we can debug, test and analyze the algorithms in controllable and repeatable environment. TOSSIM provides run-time configurable debugging output, allowing a user to examine the execution of an application from different perspectives without needing to recompile it. TinyViz is a Java-based GUI that allows you to visualize and control the simulation as it runs, inspecting debug messages, radio and UART packets, and so forth. The simulation provides several mechanisms for interacting with the network; packet traffic can be monitored, packets can be statically or dynamically injected into the network. TOSSIM is compiled by typing “make pc” in an application directory. In addition to the expected TinyOS components, a few simulator-specific files are compiled; these files provide functionality such as support for network monitoring over TCP sockets. The TOSSIM executable is build/pc/main.exe. TOSSIM has a single required parameter, the number of nodes to simulate.

By default, TOSSIM prints out all debugging information. TOSSIM output can be configured by setting the DBG environment variable in a shell. TinyViz, the Tossim user interface, provides an extensible graphical user interface for debugging, visualizing, and interacting with TOSSIM simulations of TinyOS applications. Using TinyViz, you can easily trace the execution of TinyOS apps, visualize radio messages, and manipulate the virtual position and radio connectivity of motes. In addition, TinyViz supports a simple "plugin" API that allows you to write your own

TinyViz modules to visualize data in an application-specific way, or interact with the running simulation.

2.2 Limitations of the Tossim simulator

- Every mote in the simulation runs the same TinyOS program.
  The Tossim simulator does not allow motes in the network to run different programs and communicate with each other. One way to overcome this is to write a main configuration file which initializes the mote specific sub-configurations for the respective motes based on the TOS_LOCAL_ADDRESS value.
- There is no way to directly specify connectivity between nodes in a network.
  There is no capability to directly specify connectivity between nodes in a network. By default, TOSSIM places all nodes in the simulation in a grid, where every node can listen to (is connected to) every other node in the network. To overcome this, the TinyViz user interface can be used to place the motes on it and choosing the radio model from the TinyViz plug-in.
- Tossim is focused on simulating TinyOS and its execution rather than simulating the real world. It does not capture the real world behavior.
- While Tossim simulates network behavior at bit level and simulates each individual ADC component, it does not model real world features. Instead, it provides abstractions of certain real-world phenomena (such as bit error). With tools outside the simulation itself, we can then manipulate these abstractions to implement whatever models we want to use.

CHAPTER 3 – Tython

3.1 Introduction

To overcome some of the limitations of the Tossim as explained in the Chapter 2 and to model behaviors such as mote movement, changing sensor readings and other real world phenomenon, TOSSIM provides a socket based command API for other programs to connect and issue commands to the simulator. One solution is TinyViz, a GUI which communicates with TOSSIM over the socket API. With TinyViz, we can interact with a simulation through a GUI panel by dragging motes and setting options. These actions can be difficult to reproduce exactly (e.g., dragging a mote).

Tython (or, Tinython) complements TinyViz's visualization by adding a scripting interface to TOSSIM. Users can interact with a running simulation through TinyViz, a Tython console, or both simultaneously. Tython is based on Jython, a Java implementation of the Python language. Jython makes it very easy to import and use Java classes within Python. This allows users to access the entire TinyOS Java tool chain, including packet sources, MIG-generated messages, and TinyViz.

Tython sit on top of SimDriver, a Java application that manages interactions with TOSSIM. The core of SimDriver is an event bus. Java plug-ins can connect to this event bus, can receive TOSSIM events and send TOSSIM commands. Many of the Tython abstractions are built on top of SimDriver plug-in.

SimDriver can be invoked by the following command.

 # java net.tinyos.sim.SimDriver -gui -run main.exe 10

This command will start TinyViz with the Tython console and start the simulation with 10 motes.

All TOSSIM events are sent to the SimDriver and internally distributed via the Event Bus. Periodic / future events can be implemented by inserting an event and registering a callback handler. Python scripts can register a handler to get events as well. The Tython console can be started using TinyViz GUI using the following command.

#tinyviz –run build/pc/main 10

This command starts the SimDriver, starts TinyViz GUI and connects the GUI to the simulation of 10 motes and provides a Tython console where we can import the Java classes reflected by the Jython and those of the TinyOS java tool chain. Then we can issue Tython commands on the command line which are sent to the Tython interpreter which runs concurrently with the TOSSIM simulation and interacts with it. We can import the simcore module to the Tython environment which provides the python object interface by the following command at the Tython command prompt.

*#from simcore import **

This module hides the internal complexities of the interaction of the SimDriver with the TOSSIM. This module is the core interface that is used to interact with the Tython environment. This module is in fact not Python code at all, instead, a single instance of each class of the Java net.tinyos.sim.script.reflect package is created and is bound into the simcore module.

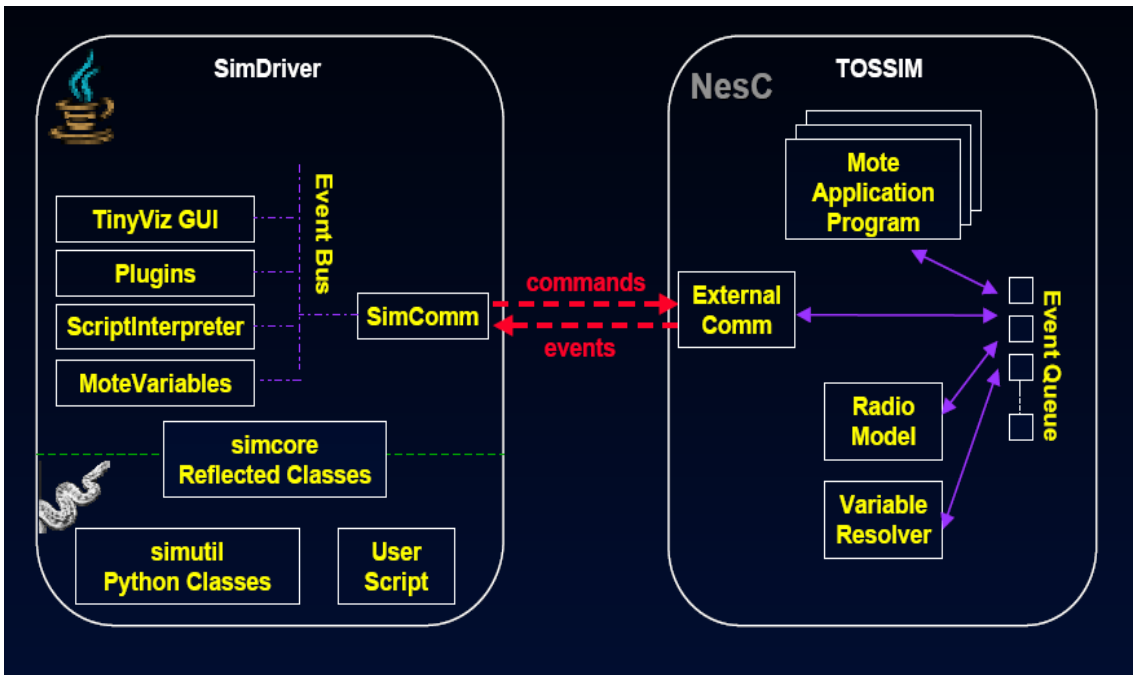

Fig. 1 – Tython Architecture

3.2 Using Tython to control the Simulation

3.2.1 Basic Commands and Modification to the Interface

When the simulation is started with TinyViz and the Tython console, the motes are assigned random locations and are displayed on the TinyViz interface and the default radio model that is used is empirical. Tossim supports two radio models, Empirical and FixedRadius. With FixedRadius we can choose either 10ft or 100ft or 1000ft as the communication range for the motes.

We can use the Tython commands to give the motes initial coordinates and to choose the radio model, so that the topology on which we want to run the simulation is ready and then give an another command to resume the simulation.

The Tython commands that can be issued to the Tossim simulation and their purpose are as follows:

- sim.pause( ) - to pause the running simulation
- sim.resume() - to resume the paused simulation
- sim.stop() - to stop the simulation
- motes[i].moveTo(x,y) – to move the mote with id i to the location (x,y)

  This is the command the Tython interface can use to place the motes on the TinyViz interface with the given coordinates. The user can provide the coordinates for each mote in XML format so that these coordinates are read and this command can be issued for each of the motes with these coordinates as the parameters.
- motes[i].turnOff( ) – to turn off the mote with id as i.

By default if we start Tython using the TinyViz, GUI it gives a command line interface to enter the commands, which are sent to the SimInterpreter internally by the SimDriver. Now as there is a need to provide to the user a graphical interface so that the user can select the commands and send them to the simulation instead of using the default command line interface, the internal java TinyOS tool chain hasis to be changed. Because of

this a change is made to the internal SimDriver class so that it waits for the GUI program to connect to it on the TCP channel on a particular port. Once we connect GUI to the TCP channel, the commands coming from the GUI can be sent to the channel and the output from the simulation can be retrieved from the channel.

3.2.2 Mobility of the Motes

Internally the movement of the motes is performed as follows. It issues the mote.moveTo() command repeatedly by calculating the next position from the destination coordinates and the amount it should get close to the destination each time, which is determined by the rate.

Sample Java Code for mote movement:

(xc,yc) – source coordinates ;

(dxc,dyc) – destination coordinates

step – increment at every period

rate is the rate of movement

os is the channel which connects to the SimInterpreter class

```
Motemove(xc,yc,dxc,dyc,step,rate)
{
    dx=dxc-xc;
    dy=dyc-yc;
    distance = Math.sqrt(((dxc-xc)*(dxc-xc))+((dyc-yc)*(dyc-yc)));
    nsteps = distance / step;
    xstep=dx/nsteps;
    ystep=dy/nsteps;
    while(yc!=dyc || xc!=dxc)
    {
            distance = Math.sqrt((dxc-xc)*(dxc-xc))+((dyc-yc)*(dyc-yc));
            if(distance < step)
            {
                    os.writeBytes("motes[0].moveTo("+dxc+","+dyc+")\n");
```

```
                xc=dxc;
                yc=dyc;
            }
            else
            {
                xc=xc+xstep;
                yc=yc+ystep;
                os.writeBytes("motes[0].moveTo("+xc+","+yc+")\n");
            }
        Thread.sleep(rate);
    }
}
```

3.2.3 Simulating External Effects

In the deployment system where we work with the real motes, whenever an external effect is sensed by the sensing component of the mote, the ADC value of the mote gets changes and the mote can respond if the observed ADC value is above some threshold which is determined by the application. For example if we have a mote with a light sensing component in it, whenever it observes the light, its ADC value gets increased and later when there is no light, ADC values gets decreased. Similarly in object tracking applications, the motes which detect the object should return high ADC values compared to the other motes.

On the other hand in simulation if we use the ADC component and call ADC.getData(), a random number is generated and is returned as ADC value. By default simulating applications which react to the external effects could not be done in TOSSIM. We have a command in Tython which set the ADC value of a mote.

comm.setADCValue(id,simTime,port,value)

where id is the mote id, simTime is the simulation time at which the value should be set, port is the port number and the value is the ADC value.

This command can be used as follows to simulate the external scenarios. Suppose if there is an XML file which has the scenario specified. There is a need to formalize the format of specifying these scenarios for various effects such as temperature detection, light sensors, object detectors, etc. Reading the scenario information the motes ADC values can be changed using the above command.

Sometimes there is a need to simulate applications where a single mote has many sensing components for example a mote can have a magnetometer to detect the objects as well as photo sensor to detect the light. To simulate these types of scenarios two ADC components should be used. As ADC is a parameterized interface we can specify the ports on which we seek to set the values as the parameter for these ADC components in the configuration file.

Example nesc code:

TestTinyVizM.ADC1 -> ADCC.ADC[1];

TestTinyVizM.ADC2 -> ADCC.ADC[2];

To set the value for first ADC port number 1 can be used and to set the value for the second ADC port number 2 can be used. In this way we can simulate the applications with multiple sensing attributes.

3.3 Sending commands to the motes

MIG is a tool that generates Java classes for TinyOS packets. The MIG tool parses C structures for TinyOS packets and builds a Java class with assessors for each of the packet fields. The Tython command used to send the message msg to mote with id moteid is as follows

comm.sendRadioMessage(moteid,simTime,msg)

CHAPTER 4 – CADENA

4.1 Modeling components and scenarios in Cadena

Cadena is an Eclipse-based extensible integrated modeling and development framework for component based Systems. Cadena provides the capability to define the modeling environments for widely used component models such as nesC a component model for sensor networks built on TinyOS. TinyOS has a component-based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks. The Cadena team chose to develop plugins to support end-to-end development for TinyOS/nesC to help support a team of developers at K-State that are currently experimenting with sensor network technologies.

We need to following steps to use Cadena: (a) Install Cadena and install the TinyOS plugins to the Eclipse/Cadena environment. (b) Create a new TinyOS project using the New TinyOS project wizard dialog. (c) Create a new TinyOS module under the current project in the module directory and choose the Style as nesC style. The TinyOS existing interfaces, components, scenarios and other libraries are modeled as Cadena interfaces, components and scenarios and provided as a zip folder TinyOSLibs.zip. (d) Import this folder in to the Cadena environment and added this to the project references list so that the existing TinyOS libraries can be used in modeling the applications. Then either new interface types can be added to the module or the existing interfaces can be imported. (e) Once all the needed interfaces are added to the module add the new component types using or providing the interfaces added to the module or import the existing components to the module.

(f) Then create a new Cadena Scenario under the Scenario directory and import the module created previously to this scenario. Components can be added to the scenario from the component types added to the module or the already existing TinyOS components can also be added to the scenarios which are added to the project as references. (g) As the last step connect the interfaces of the components in the scenario to create the complete nesC application. In the graph view of the scenario the complete

application can be seen with the components and the interface connections between these components.

4.2 Modifications made to the Cadena model

The default Cadena component model does not provide the capability to design an application with location of the components specified and does not allow connection between the interfaces of components which are at different locations. That is the global scenario cannot be modeled; instead different node level scenarios have to be designed separately which restricts the user to have the global view of the application.

The nesC style for module specification in Cadena has no property "location" for the components. There is no concept of location for the components. Since this project aims to provide the user a global view of the application which includes the multiple node level scenarios and connections between them, the nesC style was changed to include the property "location" of integer type for the component types. This property specifies where each component has to be deployed.

The other property that is added is the boolean property "observable" for the interface types to specify the commands of these interfaces as observable if it is true. The user needs to provide the implementation for these observable commands in the components so that these commands can be called from an external tool to observe the desired data or to perform some action on the node. The basic example can be to set the transmission power of the motes by sending a command to the mote. The other example could be to make the mote sleep and wake up. The observable commands can also be used to view the values some of the variables in the running network. The project also aims at designing a GUI tool which issues these commands and provide the results to the user which is discussed later in the document.

4.3 Python script to generate scenario XML file

The information in the Cadena model has to be captured in the form of an XML file which has the details about the components, their locations, the wiring information and the also the interface information like the commands, events and their parameters to generate the nesC modules files, interface files and the node level scenario files. This

XML file is very important and is used at every stage in the project framework to plug in the algorithm components based on the user needs, to generate the additional required components and wirings and also for the automation of the code generation.

The XML format is as shown below

```
<scenario>
        <name> </name>
        <component>
                <name> </name>
                <location></location>
                <type> </type>
                <input_ports/>
                <output_ports>
                        <wire>
                                <port> </port>
                                <connected_to>
                                        <wire_to> </wire_to>
                                        <comp_name> </comp_name>
                                        <location></location>
                                </connected_to>
                        </wire>
`                       </output_ports>
        </component>
        <portinfo>
                <port>
                        <name> </name>
                        <location></location>
                        <compname> </compname>
                        <periodic></periodic>
                        <observable> </observable>
                <commands_list>
                        <command>
```

```
                    <name> </name>
                    <return_type> </return_type>
                    <async></async>
                    <parameters/>
                </command>
            </commands_list>
            <events_list>
                <event>
                    <name> </name>
                    <return_type> </return_type>
                    <async> </async>
                    <parameters>
                </parameters>
                </event>
        </events_list>
    </port>
</portinfo>
</scenario>
```

As shown in the XML file the components information and the interface information along with the observable property information is captured which is later used to load these commands in the GUI. This XML file is generated using a python script which is written using the java API available for the Cadena tool in the package "edu.ksu.cis.cadena.core.specification".

For example to extract all the components information from a scenario, the pseudo code will look something as follows

```
for comp in scenario.allInstances:
    print "Scenario %s contains component %s" % (
                scenario.name,          comp.name)
    print "component type is %s" % (
                comp.type.name)
        print "component location is %d" % (
```

Integer.toString(comp.location))

To extract the interface information from a component “comp”, the pseudo code will look something as follows

```
for port in comp.ports:
    print "component %s has port %s" % (
                                comp.name,
                                port.name)
```

To run the python script right click on the graph view of the scenario and select Jython - Run Jython Script and select the python script and the XML file with all the required information is generated with the scenario name.

CHAPTER 5 – HIGHER LEVEL ABSTRACTIONS

5.1 Virtual Node

Wireless sensor network (WSN) applications exhibit a high degree of decentralization. This is particularly true of scenarios where the data reported by sensors is used to control actuators affecting the environment. Implementing this control loop in a decentralized fashion is much more complex than in mainstream, centralized applications.

Consider an application where there are many temperature sensors deployed in a field. When the average of these temperature readings of these sensors report a value higher than the threshold then the sprinklers in field should be switched on or an emergency signal should be triggered. Implementing this kind of application in a centralized manner is impractical. Thus, decentralized coordination of sensing and actuating activities increases performance but increases complexity. The available programming frameworks are too low-level and force the programmer to deal with the details of data gathering, bookkeeping, and communication, instead of focusing on the application logic. Higher-level programming abstractions are needed to deal with the complexity of decentralized sensor networks.

This project framework uses the concept of virtual nodes, a programming abstraction abstracting a set of physical nodes. The data from a set of physical sensor nodes are collected, processed according to the application specific function and provided as the single reading of a virtual node. The set of the physical nodes abstracted by the virtual node is specified by the user using the Mapping XML file which is discussed later. Using the virtual node abstraction, the user can focus on the application logic rather than the low level implementation details including the message communication and data gathering.

For example, in the following figure a set of eight physical nodes are abstracted by a virtual node. The Monitor can query the virtual node to get the temp and the virtual node gathers the data from the physical nodes apply the application specific aggregation function and provides the result to the Monitor component as a single reading. In this way the virtual node abstracts the communication details between the different physical nodes.

This also gives a scope to develop some generic functions which can be automatically plugged in based on the user requirements in place of the virtual node.

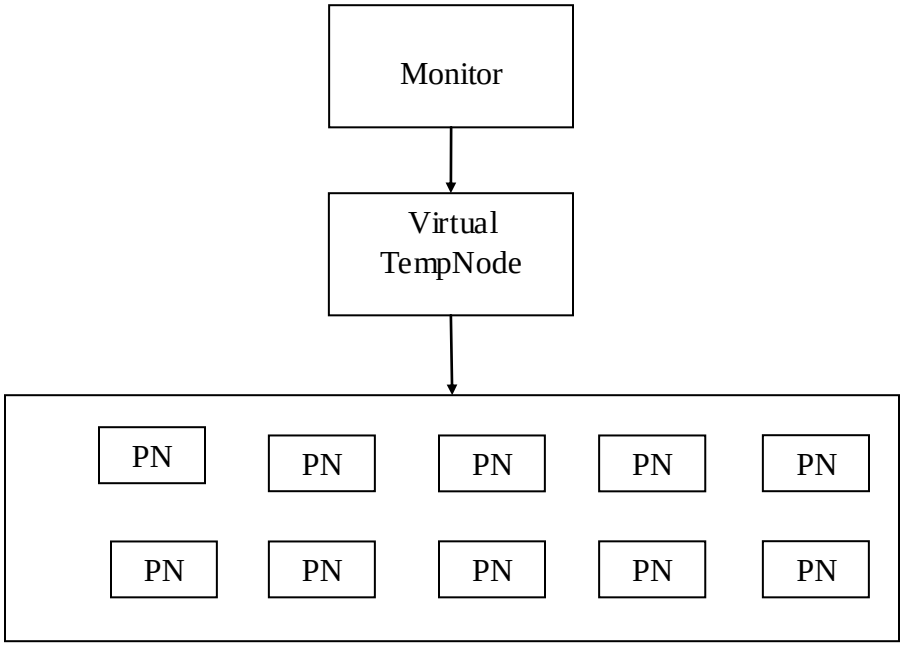

Fig. 2 – Virtual Node Model

5.2 Algorithm components (Algorithm Layer)

5.2. 1 Breadth First Search

The general use of the sensor network might be to search for a node with some search criteria and read or write variables on the searched node. This general service is provided to the user as a BFSAlgorithm.nc scenario which is in the library and automatically plugs in based on the Mapping XML file.

For example, if the virtual node is abstracting a set of physical nodes each has some data which is replicated randomly in the network and the virtual node is providing services to search for the node with the data available on it and also read that variable. Then, the BFSAlgorithm can be plugged which provides the services the user needs. It searches the network for the required attribute in a

breadth first manner and returns the nearest node available in the network satisfying the criteria. If there are multiple nodes available then the algorithm returns the first node and caches the remaining results for the future searches. This component is written so that multiple searches can simultaneously run in the network, where each one is identified by the search id issued by the base station. The BFSAlgorithm component looks as shown in the following figure

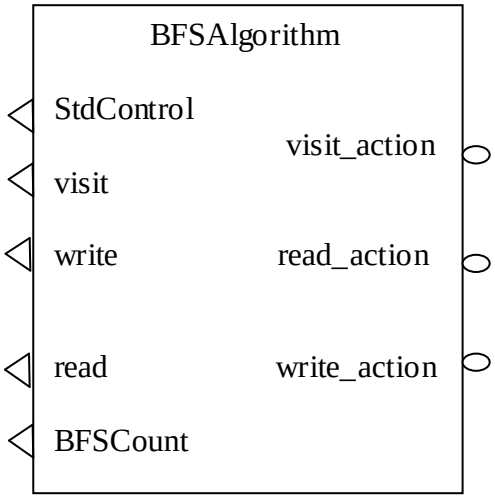

Fig. 3 – BFSAlgorithm Component 1

The visit interface has the following commands and the events

- visit – command result_t visit(char* criteria, int x1,int y1,int x2,int y2)
  visit interface take the parameters as search criteria, the bounded coordinates which represents the area that is needed to search.
- visit_reply - event result_t visit_reply(int refid,char *criteria);
  visit_reply event is signaled with the refid that is used to later access the node which satisfied the search criteria and the criteria specifying for which the search command is used.

. The read interface has the following commands and the events

- read - command result_t read(int refid,char* variable);
  read command takes tha parameters as the refid which is returned by the search call, and the variable that is to be read from the node.

- read_reply - event result_t read_reply(char* value,char* criteria);
  read_reply event is signaled when the requested data is ready and the data is returned in the string format which is then later needed to be converted to the actual data type.

The write interface has the following commands and the interfaces

- write - command result_t write(int refid,char* variable,char* value);
  write command takes the parameters as the refid, the variable name to be written and the value.
- write_reply - event result_t write_reply(char* status,char* variable);
  write_reply event is signaled with the status of the write command which may be either success or failure.

The interface BFSCount is used to get the number of messages sent at the BFS level which is used for the performance evaluation of the algorithm. visit_action interface has the command visit_action which is the action to be taken on the visit call. Similarly read_action and write_action has the commands read_action and write-action which are the actions to be taken at the physical level in respect to read and write commands.

The search algorithm used is the breadth first search algorithm which initially creates a breadth first tree on the nodes during the init process and later uses this tree to search on it. It stops if there is any positive response from the current search level. Else it searches at the next level. If no node is available then it returns -1. At every node if the search is positive it stores the address of the next hop in the path variable which is indexed by the refid and this refid is returned to the user. Thus the later reads and writes can use this refid to exactly traverse to the node using the path information.

5.2. 2 Traversal algorithms for aggregation

The other basic service that is provided as part of the project framework is generic traversal algorithm which is used to gather the data from all the physical nodes and then apply the aggregation function and return the return the result to

the user. One example of application using this might be the sprinkler system which is explained earlier in this chapter.

This service is provided as a scenario Traverse.nc which is also stored in the library and automatically plugs in based on the Mapping XML file. The Traverse component looks as shown in the following figure

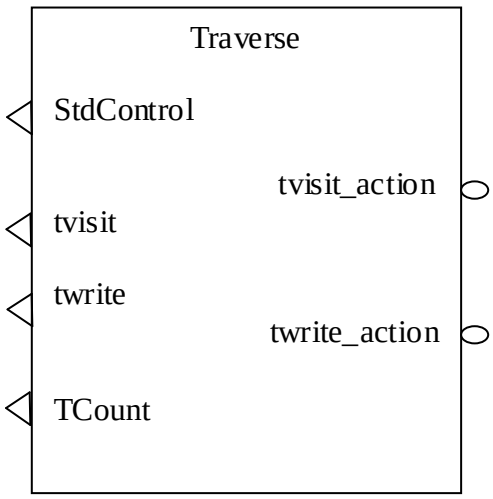

Fig. 4 – Traverse Component

The tvisit interface has the following commands and the events

- tvisit – command result_t tvisit(char* criteria, int x1,int y1,int x2,int y2)

  tvisit interface take the parameters as search criteria, the bounded coordinates which represents the area that is needed to gather the data for the aggregation.
- tvisit_reply - event event result_t tvisit_reply(int data,char *criteria);

  tvisit_reply event is signaled with the aggregated data and the criteria on which the data is aggregated.

The twrite interface has the following commands and the interfaces

- twrite - command result_t twrite(char* variable,char* value);

  twrite command takes the parameters as the variable name to be written and the value.

The interface TCount is used to get the number of messages sent at the Traverse level which is used for the performance evaluation of the algorithm. tvisit_action interface has the command tvisit_action which is the action to be taken on the tvisit call. Similarly twrite_action has the command twrite-action which is the action to be taken at the physical level in respect write command.

The traverse algorithm forms a tree rooted at the base station for the initial command and later uses this tree in its aggregations.

5.3 Mapping XML File

The Cadena component model for a nesC application can have some real components and some virtual components. The virtual components in the Cadena component model have the component names that being with "virtual_". To generate the node level scenarios these virtual components are to be collected and the associated algorithm components and the actual physical components should be plugged in. The information needed for this step is given by the user in the form of an XML file named Mapping.XML file which maps the virtual components and their interfaces to the physical components and their interfaces.

The format of the Mapping.XML file is as shown below

```
<Mapping>
    <VirtualNode>
        <Name> </Name>
        <PhysicalNode>
            <Name> </Name>
            <Searchers>
                <Command>
                    <V_int> </V_int>
                    <P_int> </P_int>
                    <V_comm> </V_comm>
                    <P_comm> </P_comm>
                    <A_type> </A_type>
                    <V_event> </V_event>
```

```
                    <P_event> </P_event>
                    <P_fun> </P_fun>
                    <Criteria> </Criteria>
                </Command>
                <Readers>
                </Readers>
                <Writers>
                    <Variable>
                        <Name> </Name>
                        <type></type>
                        <V_int> </V_int>
                        <P_int> </P_int>
                        <V_comm> </V_comm>
                        <P_comm> </P_comm>
                    </Variable>
                </Writers>
            </Searchers>
            <Aggregators>
            </Aggregators>
        </PhysicalNode>
    </VirtualNode>
</Mapping>
```

As seen from the Mapping XML file, the information about the physical node for each virtual node is captured. Each virtual node has some searchers and aggregators. Then in turn each searcher has some writers and readers and each aggregator has writers associated with it. For searchers the data in the A_type is “nearest” which specifies that the BFSAlgorithm is needed to plugged in and for the aggregators the A_type is “aggregation” which specifies the Traverse algorithm is needed to be plugged in.

Mapping file also has the details of the corresponding physical commands and the physical events of the physical node for each of the virtual commands and the virtual events of the virtual node. This information is used to generate the adapter components for the algorithms which connect the real components with the algorithm components and also to the physical components on the other end which is explained in the next section.

5.4 Adapter components

Adapter components are the bridging components that are to be generated to connect the real components with the algorithm components. These are used so that the interfaces in the algorithm components need not be changed to connect them to different physical components. Adapter components also play an important role in converting the data types to string type when they are passed to the algorithm and also to convert back them to their respective types when they are sent to the real components.

Adapter components for the algorithm components are generated in such a way that for each interface the virtual component is providing the adapter component provides that interface and from the mapping file the physical interface corresponding to this virtual interface the adapter component uses this interface. Likewise the wiring is changed accordingly.

For example consider the following model

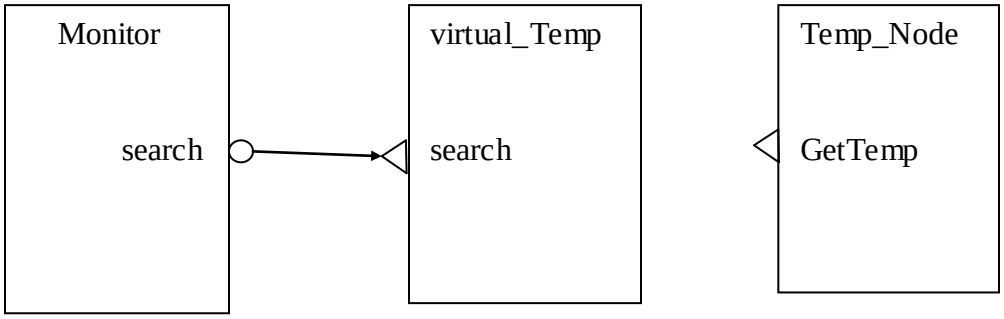

Fig. 5 – Example Model

In the above model virtual_Temp is the virtual node as its name beings with the “virtual_” keyword. Let us consider the Mapping file for this model is as shown in the following XML file

```
<Mapping>
    <VirtualNode>
        <Name>virtual_Temp </Name>
        <PhysicalNode>
            <Name>Temp_Node </Name>
            <Searchers>
                <Command>
                    <V_int>search </V_int>
                    <P_int>GetTemp </P_int>
                    <V_comm> searchtemo</V_comm>
                    <P_comm> gettemp</P_comm>
                    <A_type>nearest </A_type>
                    <V_event>search_reply </V_event>
                    <P_event> get_reply </P_event>
                    <P_fun>temp_fun </P_fun>
                    <Criteria>Temp </Criteria>
                </Command>
                <Readers>
                </Readers>
                <Writers>
                </Writers>
            </Searchers>
            <Aggregators>
            </Aggregators>
        </PhysicalNode>
    </VirtualNode>
</Mapping>
```

Since in the above Mapping XML file the A_type is nearest BFSAlgorithm component needed to plugged in and the adapter component to be generated for this component is BFSAdapter component. The final scenario that is generated after

eliminating the virtual node and plugging in the adapter, algorithm and the associated physical components is shown in the next figure.

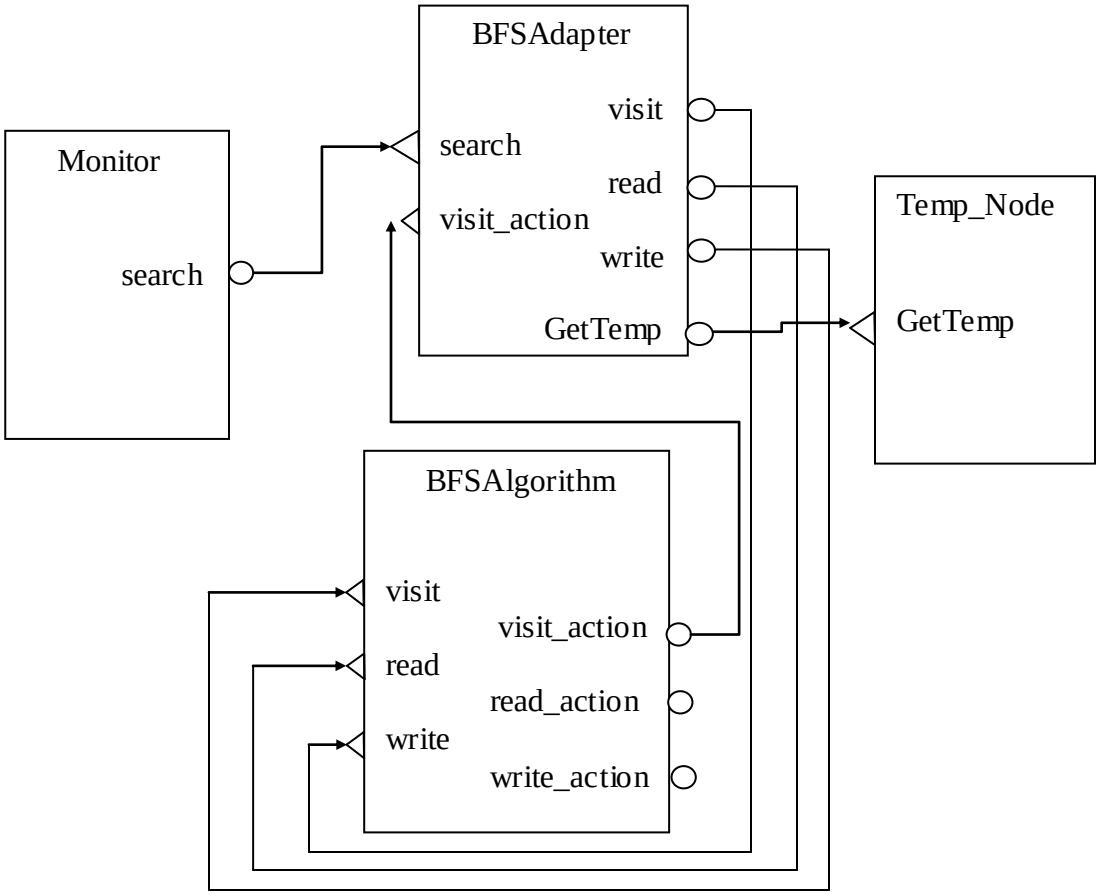

Fig. 6 – Model with adapter component

As seen from the above figure the BFSAdapter component acts as a bridge between the algorithm components and the real physical components.

CHAPTER 6 – Remote Method Invocation

6.1 Global Scenario

6.1.1 Introduction

Global Scenario in a configuration component model of the nesC application which contains components which are to be deployed at different locations in the network and the connections between the interfaces provided and used by these components. The default component model of the nesC application does not allow connecting the interfaces of the components which are to be deployed at different locations. There is no concept of the attribute location for the components in this model.

6.1.2 Modeling global scenario in Cadena

Previously, to implement the global scenario the node level scenarios are to be modeled with the components which are at one single location and connections between these components. Then the connections between the components which are at different locations are to be implemented explicitly by the user using the radio message communication between these components. The designer would not be able to look at the complete global application model, but instead able to view the unconnected node level scenarios.

This project provides a framework to the user so that the user can model the global scenario in the Cadena plug-in by adding a new attribute “location” to the component implementation in the Cadena. The location attribute specifies the physical location the component is to be deployed. The designer of the application can design the global model as if he is designing the node level model, by connecting the interfaces normally and assigning the location attribute appropriately. Thus the designer can view the global application by connecting the node level scenarios abstracted by the attribute location.

6.2 Proxy Components

Proxy components are the components that are to be generated for the remote components in the node level local scenarios when the global scenario is decomposed in to the node level scenarios. In the global scenario if an interface of a component A at location i is connected to the interface of a component B at location j, then the proxy of the component B is generated at location i and the proxy of the component A is generated at location j.

For example consider the following model

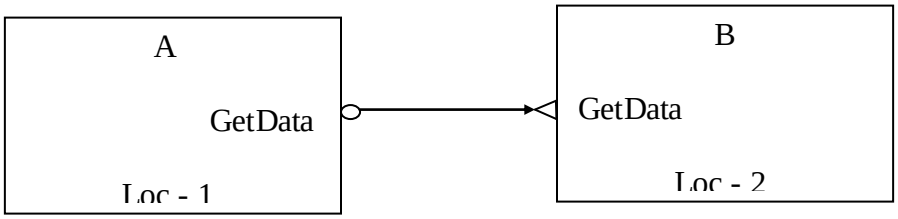

Fig. 7 – Example RMI Model

Here in this global scenario the component A is at location 1 and the component B is at location 2. A is using the GetData interface which is connected to the GetData interface provided by the component B. After generating the proxy components the application model looks as shown in the following figure

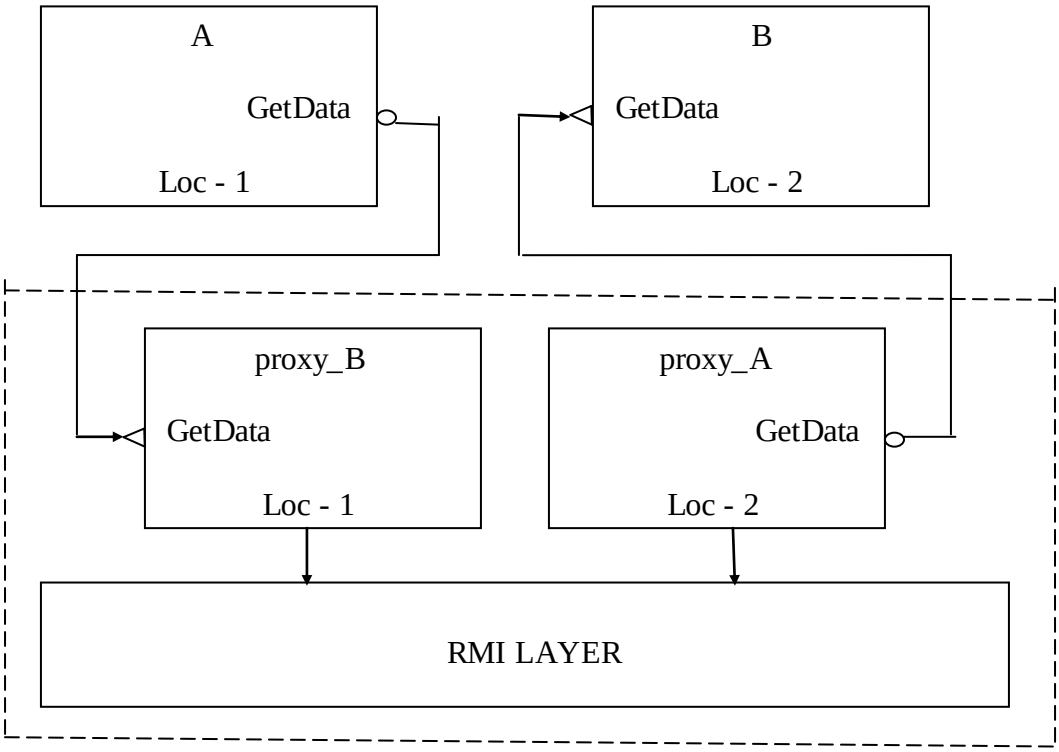

Fig. 8 – Model after RMI is plugged in

Proxy component for B "proxy_B" is generated at location 1 and proxy component for A "proxy_A" is generated at location 2. Component proxy_B provides the interface GetData which is connected to the GetData interface of the component A and component proxy_A uses interface GetData is connected to the GetData interface of component B.

The rules for generating the proxy components are:

1) If component A is using an interface connected to the remote component B then the proxy of the remote component proxy_B is generated providing the interface and the component A uses interface is connected to the provides interface of the proxy_B.
2) If component A is providing an interface connected to the remote component B then the proxy of the remote component proxy_A is generated using the interface and the component A provides interface is connected to the uses interface of the proxy_B.

Execution of the commands and events are as follows

1) If A at location 1 calls the command of the GetData interface, the proxy_B component handles this command to the RMI (Remote Method Invocation) layer,

then the RMI layer handles this command call to the proxy_A at location 2 which calls the command in the component B.

2) If B signals the event to the proxy_A at location 2, proxy_A handles this event to the RMI layer then the RMI layer handles this event to the proxy_B component at location 1 which then signals the event to the component A.

In this way proxy components along with the RMI layer are used to invoke the remote commands and to signal the remote events. Thus, this gives the designer a global view of the application abstracting the underlying communication model used for routing these commands and signals through the network.

6.3 RMI Layer Implementation

Remote Method Invocation layer is implemented as a nesC scenario RMILayer.nc which provides the interface RMIInterface and uses the interfaces PhysRcv and PhysSend provided by the physical layer.

Proxy components which use the RMI layer to execute remote commands and signal remote events have to use the RMIInterface which has the following commands and events

- RMISend - command result_t RMISend(char* msg,int destid);
  Proxy components converts the command call or a event signal along with the parameters in them as a string message and calls the RMISend() command with this string message and the destination node id where the method is to be executed as the parameters.
- RMIReceiveComplete - event result_t RMIReceiveComplete(char* m);
  RMIInterface also has an event RMIReceiveComplete which is signaled by the RMI layer to the proxy components with the string message as a parameter, which represents the command or the event when there is an RMI message received from the physical layer addressed to that particular node.

PhysRcv and PhysSend interfaces are used to receive the RMI messages from the physical layer and to send the RMI messages to the physical layer which then uses the COMM layer to send these messages over the radio communication.

RMI Layer is implemented as a reliable layer over end to end communication. To implement the reliability the protocol that is used is alternating bit protocol (ABP) which eliminates the duplicate messages and also the duplicate acknowledgements. RMI Layer has a queue to store the incoming messages from the proxy components. Two arrays are used to maintain the bitmaps one for the messages and one for the acknowledgements so that the expected bit in the messages can be saved for each of the neighbors. RMI layer also has the routing tables which has the information of the next hop for each destination in the network. When the RMI layer receives a message from the proxy component, it is enqueued and is sent over to the physical layer to send it to the next hop from the routing table information for the destination id when the medium is free. The message is resent if there is no acknowledgement received. Once the acknowledgement is received the bit is changed in the array for that neighbor. In this way the duplicate messages are dropped and also the duplicate acknowledgements are dropped by using the ABP protocol even for the acknowledgements. When the RMI layer receives a message from the physical layer it checks the destination id in the message and if the message is addressed to it, RMI layer signals the RMIReceiveComplete event with the message to the proxy component. If the destination id in the message is different from the node id then it is enqueued and sent to the physical layer to be sent to the next hop for the destination. In this way the message is routed from the source to the destination using the routing table information.

6.4 JAVA RMI

Java RMI is a mechanism that allows one to invoke a method on an object that exists in another address space. This “other address space” could be on the same machine or on a different one. For example when a process on machine A calls a method on machine B, the calling process on A is suspended, and execution of the called method takes place on machine B. Information is transported from the caller to the callee which includes the parameters and can come back in the form of the procedure result.

The idea behind RMI is that the calling procedure should not be aware that the called method is executing on a different machine. This is achieved in the following way. When the caller A calls the remote method of B, a stub identical to the remote process is created which when called, instead of executing, the method packs the parameters into a message and sends the message to the process B. Similarly on the process B, a stub identical to the process A is created and when the message arrives at the process B, the stub of process A unpacks the parameters and calls the method on B.

- Dynamic Binding

  The issue in executing RMI calls is that how the process A locates the process B. To address this issue, java RMI uses dynamic binding technique to match the processes for example clients with servers. Each process has a formal specification which contains the methods the process is providing along with the parameter information. When the process begins it sends a message to the object registry to register itself. Each process is identified by the version number, unique identifier and a handle to locate it. When a process calls the remote method, the process sends a message to the object registry asking to import the remote process. If there is any process registered with the object registry with the given version number and name, it returns the handle and the unique identifier to the calling process stub.

- Comparison of Java RMI and RMI provided by the project framework

  The RMI service provided in this project framework uses the similar procedure as it is in the Java RMI in generating the stubs for the remote processes. It also uses the same technique in packing the method call along with the parameters in a message and sending it over to the remote process for execution.

  The main difference between both of them is the way the remote processes are located for sending the messages. While the Java RMI as explained above uses the object registry so that the remote processes can register themselves, there is no concept of the object registry in the RMI framework used in this project. Instead it is assumed that a shortest route algorithm is run on the network so as to populate the routing tables present at each and every location

in the network. Whenever a node calls a remote command the information present in this routing table is used to identify the remote process and route the command to it.

CHAPTER 7 – GRAPHICAL USER INTERFACE LAYER

7.1 Introduction

As mentioned in the Chapter 4, the Cadena model is modified so that the interfaces have an additional boolean attribute “observable” to identify these interfaces as the observable interfaces if it is set to true so that the commands of these interfaces can be called from the GUI to view the data interested to the user. In this way the commands that the user want to issue from the GUI are designed as the interfaces to the components with the observable attribute as true for these interfaces. The other application interfaces has the observable attribute either set to false or can be left as default value.

7.2 GUI Components

GUI components are the components which are to be generated for the components that has the observable property as true for any of the interfaces in its provides list. If a component A at location i have a interface GetX in its provides list with the observable property as true for this interface, then the gui component with respect to the component A “gui_A” has to be generated with the interface GetX added to its uses list.

For example consider the following component

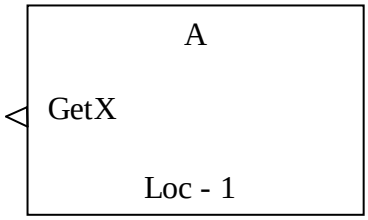

Fig. 9 – Example GUI component

Here the component A at location 1 provides the interface GetX with the observable property as true. After generating the GUI component the application model looks as shown in the following figure

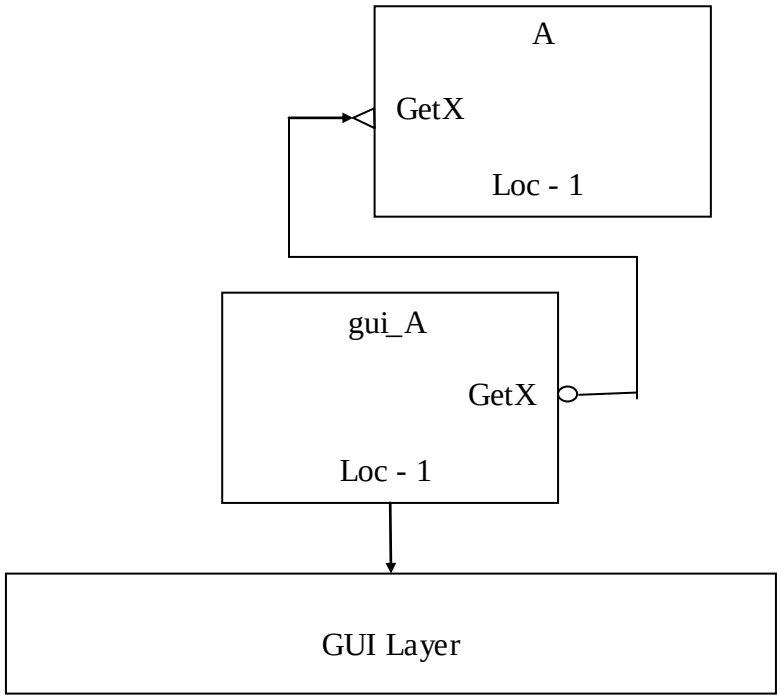

Fig. 10 – Model after GUI is plugged in

GUI component for the component A "gui_A" is generated at location 1 with the interface GetX in its uses list and this interface is connected to the GetX interface provided by the component A.

The rules for generating gui components are as follows

1) If a component A is providing an observable interface, then the gui component "gui_A" is generated for that component which uses that interface and is connected to the interface provided by A.
2) The gui component generated should use the interface GUIInterface and is connected to the GUI Layer, which is used to send messages to the GUI and receive message from the GUI through UARTControl.

In this way the gui components along with the GUI layer are used to get the commands from the GUI, execute the commands and send the results back to the GUI to

display for the user in response to these commands. Thus giving the designer a way to specify the commands in the application model and executing the commands and returning the interested data to the user using the abstraction of underlying GUI layer and the gui components.

7.3 GUI Layer Implementation

Graphical User Interface layer is implemented as a nesC scenario GUILayer.nc which provides the interface GUIInterface and uses the interfaces PhysRcv and PhysSend provided by the physical layer.

GUI components which use the GUI layer to execute the commands from the user and send responses back to the user have to use the GUIInterface which have the following commands and events

- GUISend - command result_t GUISend(char* msg);

  GUI components converts the command call or a event signal along with the parameters in them as a string message and calls the GUISend() command with this string message as the parameter.

- GUIReceiveComplete - event result_t GUIReceiveComplete(char* m);

  GUIInterface also has an event GUIReceiveComplete which is signaled by the GUI layer to the gui components with the string message as a parameter, which represents the command along with the parameters converted to string type when there is an GUI message received from the physical layer addressed to that particular node.

PhysRcv and PhysSend interfaces are used to receive the GUI messages from the physical layer and to send the GUI messages to the physical layer which then uses the UARTControl layer to send these messages over the UART back to the GUI.

GUI layer do not process any message, whenever it gets a message from the physical layer it simply forwards it to the gui component and whenever it gets a message from the gui component it forwards it to the physical layer. It acts as a bridge between the gui components and the physical layer. Once the gui components gets the message from the GUI layer, the message is decoded to extract the interface name, command name and the parameters. Then the command in that interface is to

be called using the parameters converted back from string type to their respective data types.

CHAPTER 8 – PHYSICAL LAYER

8.1 Introduction

The physical layer is the low level layer and the most important one in the framework. All the upper layers GUILayer, RMILayer, BFSAlgorithm and Traverse algorithm layers use physical layer to send and receive messages over the radio communication and the UART Control.

8.2 Implementation

Physical Layer is implemented as a nesC scenario PhysLayer.nc which provides the interfaces PhysSend and PhysRcv and uses the interfaces SendMsg, ReceiveMsg provided by the GenericComm and UARTComm layers.

All the upper layers in the framework which use PhysSend interface to send the radio messages has the following commands and events

- send - command result_t send(char* message,char dcomp,int dest);
  send command is called with the parameters message string, type of the message and the destination id whenever any layer wants to send a message. The dcomp parameter is used to de multiplex the message upon the reception of the message on the destination such that the messages sent by one layer are provided to the same layer on the destination. The dest parameter is used to identify where the message is needed to be delivered. If it is -1 then message is broadcasted, or if it is -2 then the message is sent over the UART and if it is anything else then the message is sent to that particular id. The dest parameter value -2 is used when the GUILayer wants to send the replies of the commands to the GUI on the PC. The dcomp character along with the source node id added to the message string and is sent using the send command of the SendMSg interface.
- sendDone - event result_t sendDone();

sendDone event is signaled by the physical layer when the physical layer gets the event sendDone from the Comm layer indicating that the message transmission is completed by the Comm layer.

All the upper layers in the framework which use PhysRcv interface to receive the radio messages has the following event

- receive - event result_t receive(char* message,char dcomp);

  Physical Layer signals the receive event when it receives the message from the Comm layer. It extracts the dcomp character from the received message and include this character as a parameter to the receive event. Upon reception of this event at the upper layer, the dcomp character is used to identify if this message is targeted for it. In this case if it is addressed to that layer then the message is processed else the message is ignored. The dcomp character for the GUILayer is 'G', for the RMILayer is 'R', for the BFSAlgorithm is 'B' and for the Traverse is 'T'.

Thus the physical layer commands can be used by the upper layers to send and receive the messages over the radio communication and also to send the message to the UART and receive from UART control abstracting the underlying Comm layer details.

The level of the physical layer in the whole application framework is shown in the following diagram.

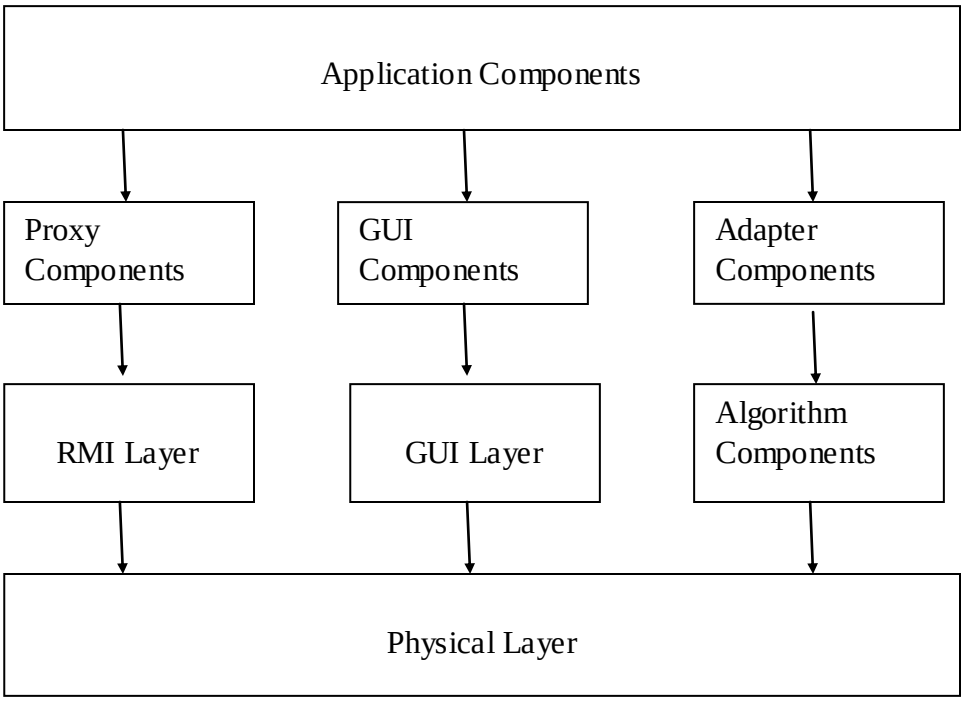

Fig. 11 – Complete Architecture

CHAPTER 9 – CODE GENERATION

9.1 Loading the Scenario XML file

Once the global scenario is modeled in Cadena tool, the user can use the python script and generate the scenario XML file which contains the complete details of the application model in XML format. The details include the components, their locations, wiring information and also the regarding the ports their names, commands, events and their parameters and the observable property information.

This scenario XML file can be used along with the user provided abstractions to generate the node level scenarios with the make files so that they are ready to be deployed on the motes.

The initial step in generating the code is to load the scenario XML file, parse it and save the information in this XML file as java objects so that they are used in the later stages.

The structure of the java objects are as follows

- Component(String comp_name, LinkedList uses, LinkedList provides, int location, String type)
    - comp_name – name of the component
    - uses – the uses interface list and their wiring information (list of wiring class objects)
    - provides – the provides interface list and their wiring (list of wiring calss objects)
    - location – location of the component
    - type – base type of the component
- wiring(String input_interface_name, LinkedList connections)
    - inpit_interface_name – the name of the interface
    - connections – the list of other components to which this interface is connected (list of interfacedef class objects)
- interfacedef(String comp_name, String int_name, int location)

    - comp_name – name of the component to which interface is connected
    - int_name – name of the interface
    - location – location of the component to which interface is connected
- Scenario(int location, LinkedList components)
    - location – location of the scenario
    - components – list of components at that location
- interfaceclass(String name, String comp_name, **boolean** observable, LinkedList commands, LinkedList events)
    - name – name of the interface
    - comp_name – interface the component belongs to
    - observable – attribute which specify it as a GUI command
    - commands – list of commands (list of command class objects)
    - events – list of events (list of command class objects)
- Command (String n, String rt, boolean a, LinkedList params)
    - n – command name
    - a – async information of the command
    - params – list of parameters and their types

9.2 Loading the Mapping Xml File

As explained in Chapter 5, the Cadena component model consists of some virtual components and some real components. The higher abstractions provided by the user, which maps the interfaces provided by the virtual components to those interfaces of the physical components are given in the format of an XML file which is called Mapping XML file. The next step in the code generation is to load the Mapping XML file, parse it and store it internally as java objects so that it is used in the later stages.

The structure of the java objects are as follows

- VirtualNode(String name, LinkedList physicalnodes)

- name – name of the virtual node
- physicalnodes – list of the physical nodes under this virtual node

- PhysicalNode(String name, LinkedList searchers, LinkedList aggregators)
  - name – name of the physical node
  - searchers – list of searchers under this physical node
  - aggregators – list of aggregators under this physical node
- Searcher(String vir_int, String phys_int, String vir_comm, String phys_comm, String acc_type, String vir_event, String phys_event, String function, String criteria, LinkedList readers, LinkedList writers)
  - vir_int – name of the virtual interface
  - phys_int – name of the physical interface
  - vir_comm – name of the virtual command
  - phys_comm – name of the physical command
  - acc_type – accessor type to determine the algorithm needed to plug-in
  - vir_event – name of the virtual event
  - phys_event – name of the physical event
  - function – function used at the adapter level
  - criteria – criteria for the search
  - readers – list of readers associated with this search
  - writers – list of writers associated with this search
- Aggregator (String vir_int, String phys_int, String vir_comm, String phys_comm, String acc_type, String vir_event, String phys_event, String function, String criteria, LinkedList writers)
  - vir_int – name of the virtual interface
  - phys_int – name of the physical interface
  - vir_comm – name of the virtual command
  - phys_comm – name of the physical command

    - acc_type – accessor type to determine the algorithm needed to plug-in
    - vir_event – name of the virtual event
    - phys_event – name of the physical event
    - function – function used at the adapter level
    - criteria – criteria for the search
    - writers – list of writers associated with this search
- Reader(String name, String type, String vir_int, String phys_int, String vir_comm, String phys_comm, String vir_event, String phys_event)
    - name – name of the reader
    - type – type of the variable to be read
    - vir_int – name of the virtual interface
    - phys_int – name of the physical interface
    - vir_comm – name of the virtual command
    - phys_comm – name of the physical command
    - vir_event – name of the virtual event
    - phys_event – name of the physical event
- Writer(String name, String type, String vir_int, String phys_int, String vir_comm, String phys_comm)
    - name – name of the writer
    - type – type of the variable to be written
    - vir_int – name of the virtual interface
    - phys_int – name of the physical interface
    - vir_comm – name of the virtual command
    - phys_comm – name of the physical command

9.3 Elimination of Virtual Nodes

Once the information about the components and the mapping information of the virtual nodes to the physical nodes is obtained the next step is to replace the virtual nodes with their respective physical nodes. This step also deals with the

addition of new components which includes the algorithm components and their adapter components to the total components list.

The pseudo code for the elimination of the virtual nodes is given below

- *Scan Scenario XML file and form a list of virtual components as VirtualComps.*
- *Scan Mapping XML file and form a list of mapped virtual components as MappedComps*
- *For each virtual compoenent vnode in VirtualComps*

For each mapped component mnode in MappedComps

If vnode.name == mnode.name

For each physical node pnode in vnode

For each searcher search in pnode

If pnode.acc_type == "nearest"

Plug BFS Algo

Add appropriate wirings to BFSAdapter components

For each readr in search

Add wirings

For each writer in search

Add wirings

For each aggregator aggregate in pnode

If pnode.acc_type == "aggregation"

Plug Traverse Algo

Add appropriate wirings to TraverseAdapter components

For each writer in aggregate

Add wirings

After plugging in the required algorithm components and adding the appropriate adapter components and their wirings, the virtual components that are the

components whose name starts with “virtual_” can be removed from the total list of the components.

9.4 Generation of node level scenarios

As explained in Chapter 6, the global scenario has components having different values for their location attribute and these components are connected to each other thus giving the user a global view of the application.

The next step in the code generation is to generate the node level scenario wirings by introducing the proxy components and connecting the proxy components using the RMI layer.

The pseudo code for generating the node level scenarios is given below

- *For each component comp in total components*

  For each wiring port in comp.uses

  For each connection conn in port.connections

  If conn.location == comp.location

  Local wiring no need of any change

  Else

  Create component “proxy_”+conn.compname at location comp.location

  Modify conn to connect to this new component

  For the component conn.compname remove this connection from its provides list

  For each wiring port in comp.provides

  For each connection conn in port.connections

  If conn.location == comp.location

  Local wiring no need of any change

  Else

  Create component “proxy_”+conn.compname at location comp.location

  Modify conn to connect to this new component

For the component conn.compname remove this connection from its uses list

- *Add the wirings from the proxy components to the RMI Layer components*

Once the proxy components are generated then the components which are at one location can be collected and group them as local scenarios. Later the XML files for these local scenarios can be generated.

9.5 Generation of GUI Components

The next step is to generate the GUI Components for those components which have any of the interface with the observable property as true in its provides list. Observable interfaces are the interfaces with commands that can be called from an external graphical user interface.

The pseudo code for generating the GUI Components is given below

- *For each scenario scen in total scenarios*

For each component comp in scen.components

For each wiring in comp.provides

For each interface port in total ports

If port.name = wiring.name

If port.obervable = = true

Add port to observeinterfaces list

If observeinterfaces.size() >0

Create component "gui_"+comp.name

For each interface in observeinterfaces

Add wiring to gui component uses list

Add wiring to comp provides list

In this way the gui components for the components that have observable interfaces are generated. Whenever GUI issues a command, the gui component respective to the actual component receives this command from the UART control

and parses the command to get the interface name and the command name. Then the command in that interface is called by the gui component. Similarly if any event is signaled to the gui component, it sends a message back to the GUI through UART control.

9.6 Generation of code for components and interfaces

In this step the configuration files for every scenario and the module files for each of the components and also the interface files for the interface used or provided by these components are generated. Here in this step the code is generated for the commands and the events of the proxy components, gui components and also for the adapter components.

9.7 Generation of Make files

Finally the make file is to be generated for every node level scenario with the main component as the main configuration file for that scenario and also setting PFlags to include the library where the library components are stored. Files specific to each scenario are generated in its own folder and are ready to be deployed on the motes.

CHAPTER 10 - GUI FOR DATA VISUALIZATION

10.1 Introduction

The concept of having a GUI for the sensor network is to control the running sensor network applications and to visualize the data interested to the user from the network by issuing the commands from the GUI to the network and get the results back from the network to the GUI in both simulation mode and also in the deployed system

Thus, the GUI is needed to be initialized as shown below which can interact bidirectional with both the simulation and the deployed system using respective interfaces. The Simulation interface here would be Tython environment using which we can issue the commands to the network. In Tossim, Serial Forwarder can receive the messages only from the more with TOS_LOCAL_ADDRESS as 0 which is the base station of the network.

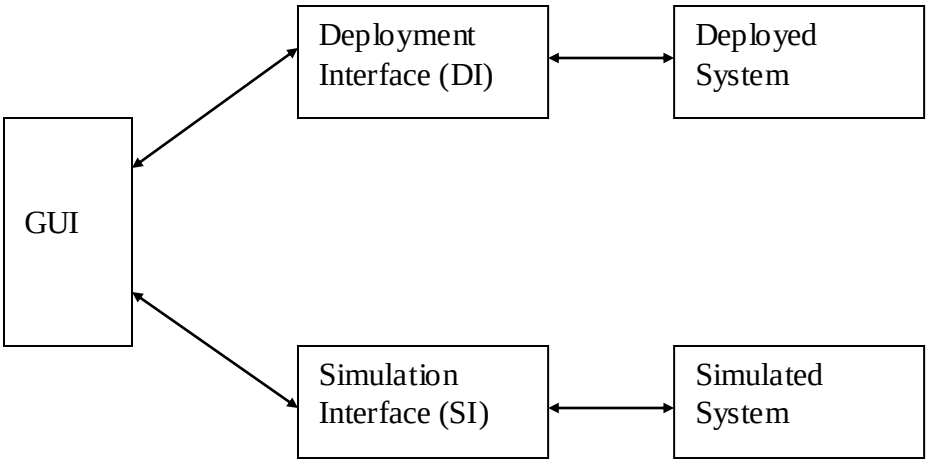

Fig. 12 – GUI Concept

a specific port and using the serial forwarder tool to send the messages and receive the messages from the network to the GUI.

10.2 Loading the topology XML file

Before the simulation is started or the GUI connects to the deployed network, the topology information should be loaded by the GUI so that the GUI displays the same topology as it in the deployment or simulation.

The format of this XML file is as follows

```
<topology>
    <mote>
        <id>0</id>
        <x>22</x>
        <y>22</y>
    </mote>
    <mote>
        <id>1</id>
        <x>28</x>
        <y>28</y>
    </mote>
</topology>
```

The above file tells the GUI to place the mote with id 0 at the location (22, 22) and to place the mote with id 1 at the location (28, 28). In Simulation the Tython command is used to place the motes so that the same topology can be seen in the TinyViz interface.

10.3 Loading the Scenario XML file

To send the commands to the network the GUI should know what commands are to be sent, the parameters associated with these commands and the responses for these commands. This information can be embedded in the application model when the user models the application in Cadena by setting the observable property of some of the interfaces as true. As specified in the previous chapters this information is extracted using the python script and captured in a XML file.

The GUI should load this XML file, parse it and identify the commands that are to be issued for each mote.

The GUI with the motes placed on it and the commands loaded from the scenario XML file is as shown below

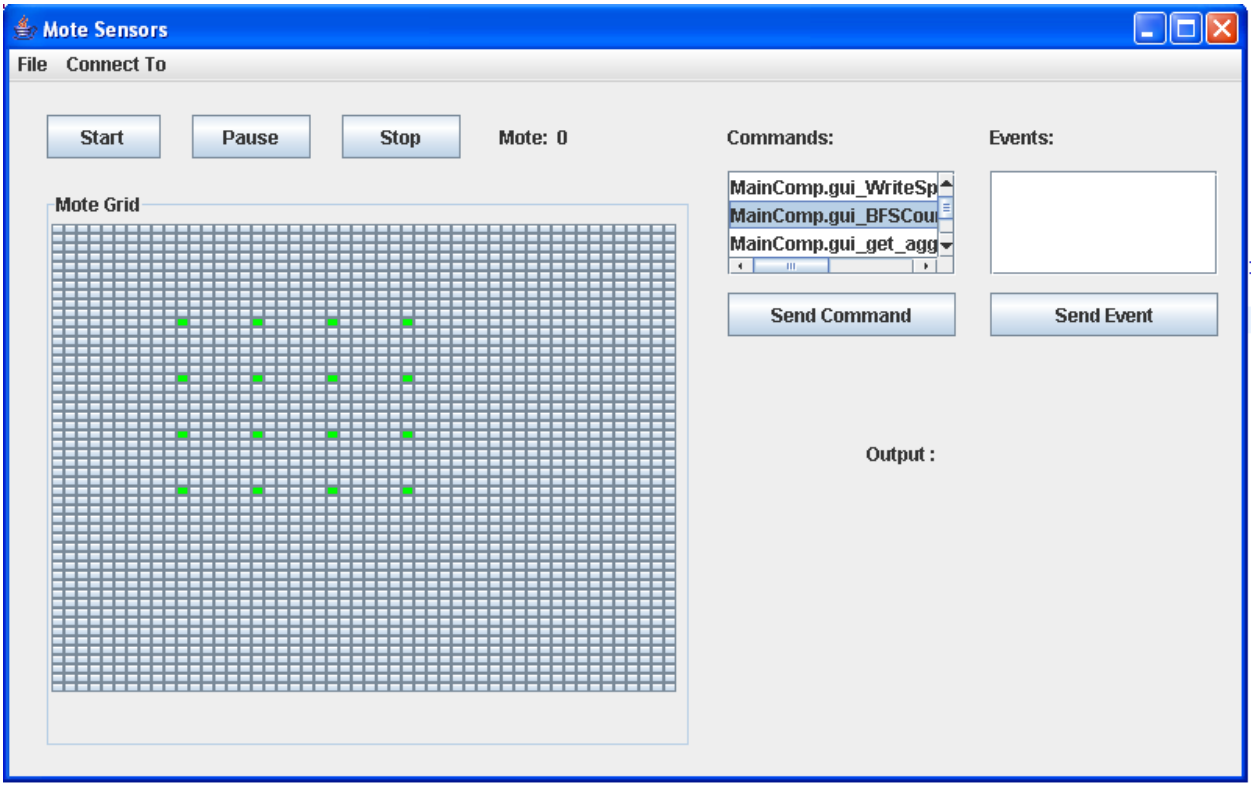

Fig. 13 – GUI Snapshot 1

It has a menu File with Load Topology and Load Observable Commands submenus to load both the topology XML file and the Scenario XML file respectively. There is also a Commands List on the GUI that is populated with the commands that can be issued when a specific mote is selected on the GUI. Then the command can be sent by selecting the Send Command button.

There are also buttons like Start, Pause and Stop which are used to start or resume the simulation, pause the simulation and stop the simulation respectively.

10.4 Connect to the Simulation or Deployment

Once the GUI is loaded with the topology and the scenario information, the GUI can either be connected to Simulation or Deployment by using the Connect Menu and selecting either Simulation or Deployment submenu.

If the Simulation option is selected then the GUI connects to the TCP channel of the SimDriver class so that the commands can be written to this channel and the output can be extracted from this channel. The Serial Forwarder is started with sf@tossim-serial MoteComm variable to receive the messages from the base station through UART control.

If the Deployment option is selected then the GUI prompts the user to enter the post number so that the GUI can be connected to any specific mote using serial@COM+"portnum:"+57600 MoteCom variable where 57600 is the baud rate for telosb motes

Once the GUI is either connected to the Simulation or Deployment, the user can select the commands from the list and can issue them. If there are any parameters for the commands the GUI prompts the user to enter the parameters. If the GUI received response for any of its commands they are displayed to the user in a new window. The message format that the GUI sends is the "CompName.InterfaceName.CommandName(parameter list)". Parameter list is formed by converting all the parameters of the command to string type. The results are received in the same format. Once the GUI gets the output it can used the interface name and command name to identify the command for which the output is obtained and can show the appropriate prompt message to the user.

CHAPTER 11 – EXAMPLES

11.1 Parking Lot

In this example the concept of the virtual node is demonstrated which has some searchers with the access type as "nearest" so that the BFSAlgorithm can be plugged in. Assume that there is a sensor node at every parking lot in a parking area which maintains the status of the availability of the parking lot. When a car approaches the client which is the GUI running on a PC, it induces the search command to the virtual node. Then the whole network is searched in a breadth first manner and the reference to the nearest node available is returned to the user. The user can then use this reference to update the status of this parking lot to be reserved and can issue additional read requests on the attributes of the parking lot.

The Cadena component model for this application looks as shown below

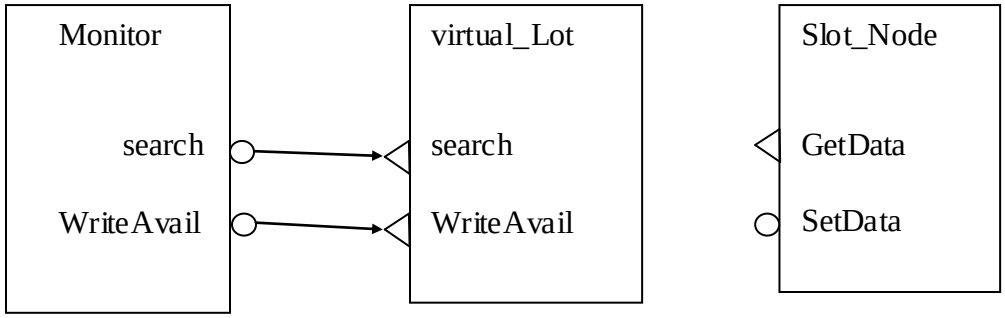

Fig. 14 – Parking Lot Model

In the above model virtualLot is the virtual node as its name beings with the "virtual_" keyword. Let us consider the Mapping file for this model is as shown in the following XML file

```
<Mapping>
    <VirtualNode>
        <Name>virtual_Lot </Name>
        <PhysicalNode>
            <Name>Slot_Node </Name>
            <Searchers>
```

```
<Command>
    <V_int>search </V_int>
    <P_int>GetData </P_int>
    <V_comm> search</V_comm>
    <P_comm> getdata</P_comm>
    <A_type>nearest </A_type>
    <V_event>search_reply </V_event>
    <P_event> getdata_reply </P_event>
    <P_fun>avail_fun </P_fun>
    <Criteria>Slot </Criteria>
</Command>
<Readers>
</Readers>
<Writers>
    <Variable>
        <Name>avail</Name>
        <type>bool</type>
        <V_int>WriteAvail</V_int>
        <P_int>Setdata</P_int>
        <V_comm>writeavail</V_comm>
        <P_comm>set</P_comm>
    </Variable>
</Writers>
</Searchers>
<Aggregators>
</Aggregators>
</PhysicalNode>
</VirtualNode>
</Mapping>
```

Since in the above Mapping XML file the A_type is nearest, BFSAlgorithm component needed to plugged in and the adapter component to be generated for this

component is BFSAdapter component. The final scenario that is generated after eliminating the virtual node and plugging in the adapter, algorithm and the associated physical components is shown in the next figure.

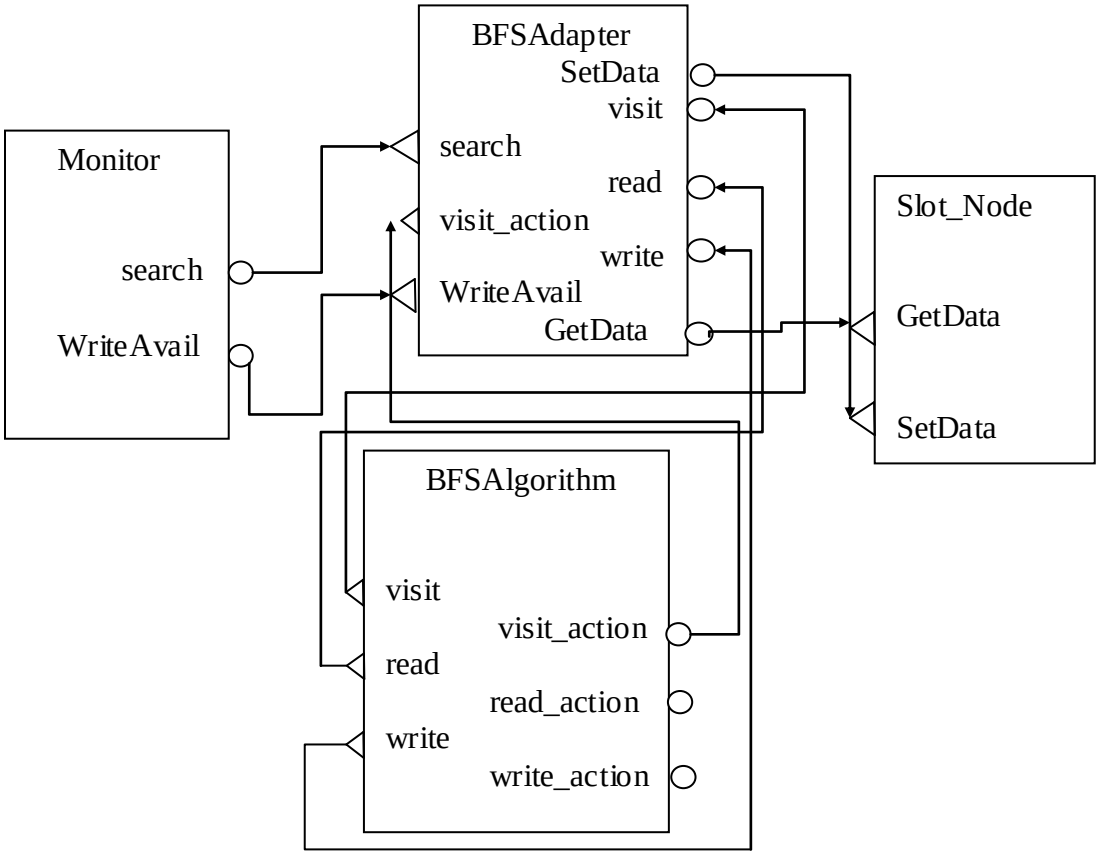

Fig. 15 – Parking Lot model deployment

When the Monitor component calls the method search, the BFSAdapter calls the visit of the BFSAlgorithm with the given search criteria, The BFSAlgorithm then calls the visit_action command in the BFSAdapter, which in turn calls the GetData command. The return event of this call is sent to back to the adapter which processes the result using the function mentioned in the search mapping file and returns either true or false to the BFSAlgorithm. If this is true its id is stores in the path and a reference to this path is sent to the user. Else the BFSAlgorithm sends a message to all its neighbors to search at next level. In this way the whole network is searched by the BFSAlgorithm until a positive response is

received at any search level. Thus the parking lots are abstracted by a single virtual lot and can respond to the user queries.

11.2 Kitchen Application

In this example, the usage of the remote method invocation layer and the concept of the proxy nodes are demonstrated. Consider an application where there is a need to monitor a kitchen so that an alarm can be fired if the stove is on and there is no one in the kitchen room. There are three scenarios that are needed. One is to sense the temperature of the stove to determine if it is on. Second one is to sense the pressure on the floor to determine if any is monitoring the stove and third scenario is to fire an alarm based on the sensor values of the first and the second scenario.

The Cadena application model for this application might look something as follows

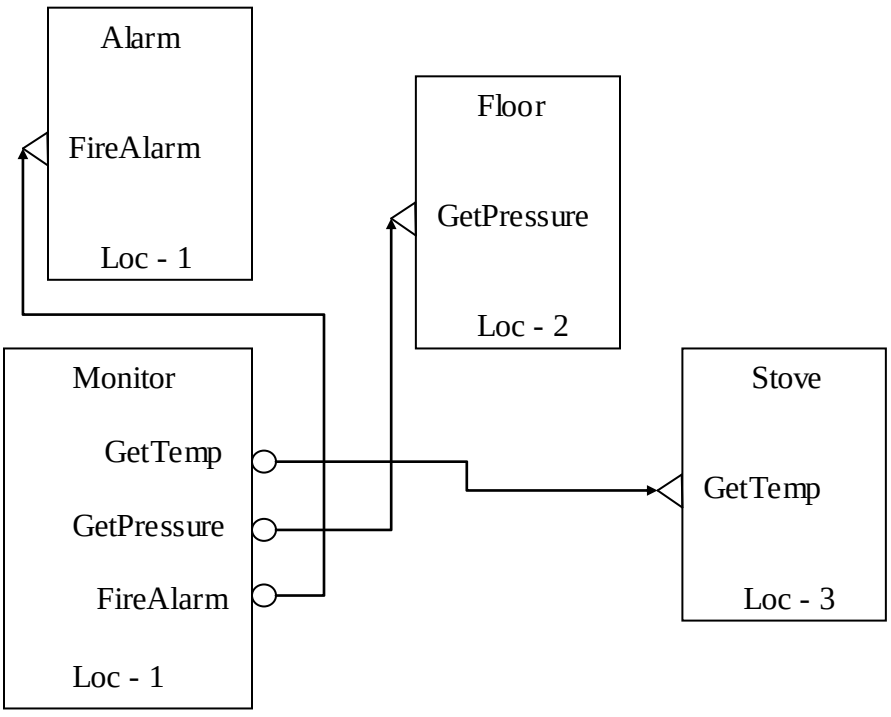

Fig. 16 – RMI Example Model

location1, Floor component is deployed at location 2 and the Stove component is deployed at location 3.

When this model is processed by the tool which generates the node level scenarios, three separate scenarios are generated each to be deployed at its specific location. The node level scenarios look something as follows

Scenario 1

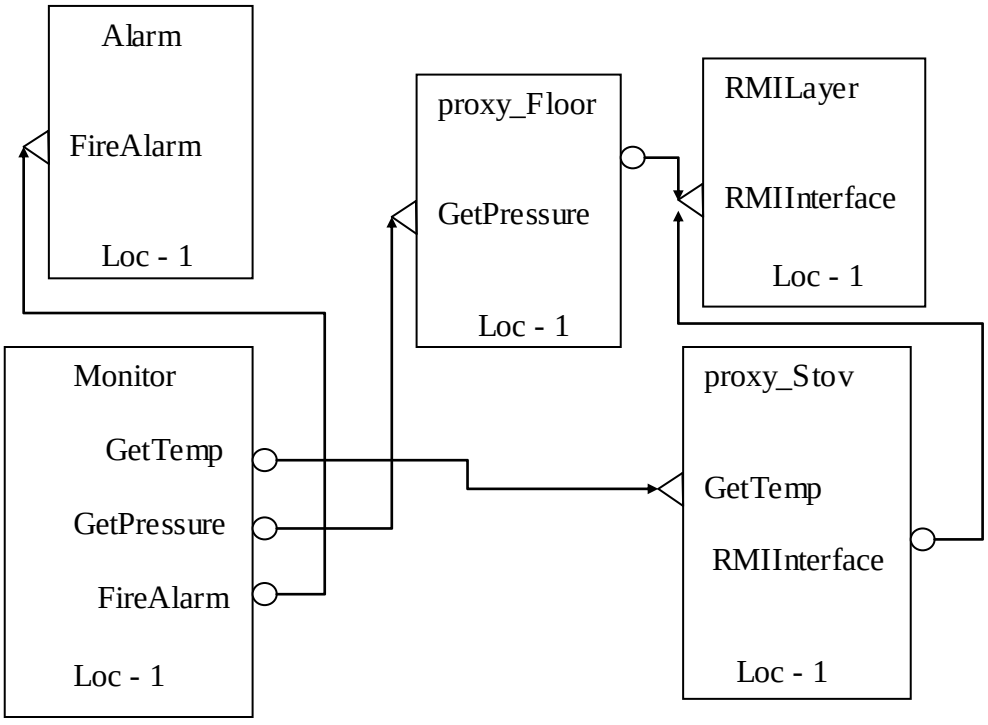

Fig. 17 – Scenario 0 Model

Scenario 2

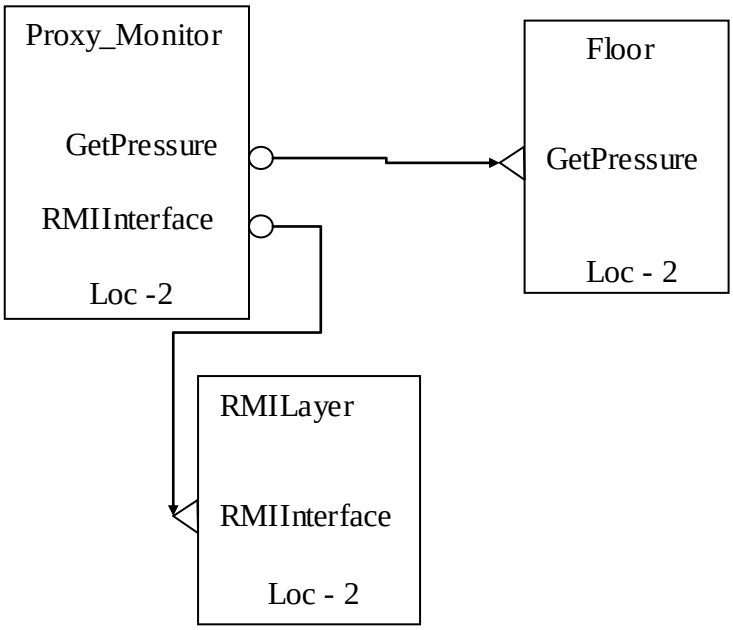

Fig. 18 – Scenario 1 Model

Scenario 3

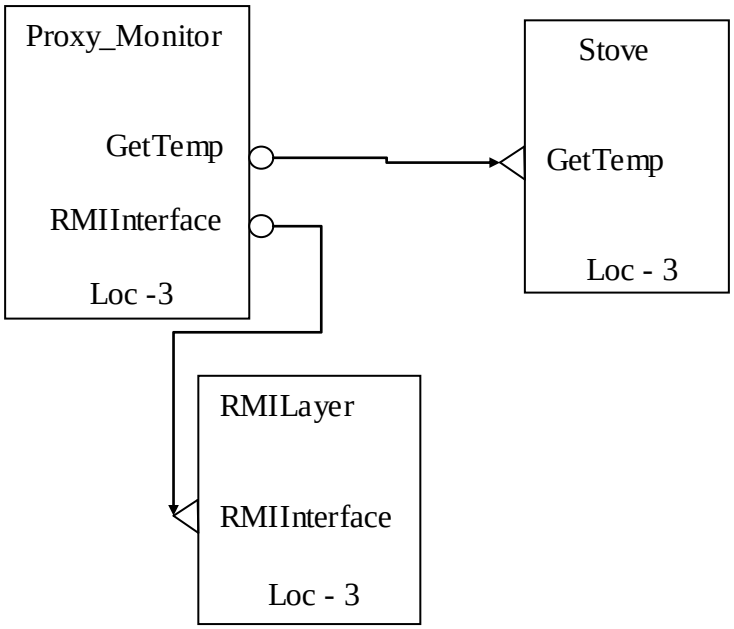

Fig. 19 – Scenario 2 Model 1

In this way the node level scenarios are generated which can be deployed on three different motes. The underlying physical topology is known to the RMI layer in the form of the routing tables which can use this information to route the command calls and event calls to the specific motes using the radio communication.

CHAPTER 12 – PERFORMANCE RESULTS

12.1 Performance

- BFSAlgorithm

A Cadena model is developed for the parking lot example as explained in the Chapter 11. Then the XML for this scenario is generated using the python script and the node level scenario files are generated using the XMLtoNesC java tool. This scenario is deployed on a test bed with 16 telosb motes arranged in a 4 by 4 grid topology. Test bed has a central USB hub which connects to each of the telosb motes on the bed and also supplies power for them. To calculate the number of messages sent by each of the mote, the GUI needs to connect to each of the motes on their particular port number and issue the command to retrieve the result. So an extra interface is added to the model with observable attributes as true to get this result.

The algorithm is tested on both simulation and deployment and the results are as follows

No of messages	Tree Formation	1st search	2nd search	3rd search	4th search
Simulation	133	32	28	22	20
Deployment	226	111	92	73	44

Fig.20 - Search performance

The observations that can drawn for the above table are that the number of messages used for the subsequent searches are getting decreased as some of the internal nodes are mark closed in the previous searches if there are no positive responses from any of their children. And also the number of messages used is more on the deployment compared to the simulation which explains the fact that more messages are lost because of the interface on the test bed compared to the simulation.

The read and write operations results are as follows

No of messages	Read	Write
Simulation	10	7
Deployment	13	8

Fig. 21 – BFS Operation performance

The results for when two simultaneous searches are issued are as follows

No of messages	2 searches
Simulation	79
Deployment	146

Fig. 22 – BFS Multiple search performance

In the above case if the algorithm is modified so that for multiple simultaneous searches having the same criteria, only one search is allowed to go and the rest are blocked for the first search the number of messages for the rest of the subsequent searches are almost zero as the first search result is used to return for these search requests.

Consider the following topology

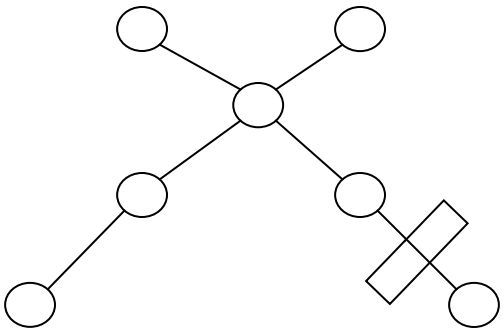

Fig. 23 – Topology Deployed on Field

The search and the write operations performance on this topology where the distance between each of the motes is 10 ft is as shown in the following table

Noofmessages	TF	search	write	search	write	search	write	search	write	search	write
Deployment	133	9	4	6	6	3	6	56	12	30	13

Fig. 24 – Search performance on the Real Field

The observations that can be drawn are that the number of messages are decreased for the subsequent searches at the first level and again follows the same pattern for the searches at the second level. As there is a wall between two of the motes, more message loss is observed between these motes.

- Aggregation

To test the aggregation algorithm sprinkler application is modeled in Cadena and deployed on the test bed in the same way as in explained for the BFS Algorithm.

The results observed for the different operations on the aggregation algorithm are as follows

No of messages	Avg	Max	Min	Write
Simulation	63	51	47	24
Deployment	76	63	58	34

Fig. 25 – Aggregation Operation performance

The observations that can be drawn from the above table are that for the first aggregation operation more number of messages is used because this operation constructs the tree on the physical topology. Then the later operations use this tree.

- RMI Layer

The RMI Layer is tested on different scenarios. The first one is for the single hop; second one is for the 2 hops and third one with both single hop and two hops and with multiple components.

The number of messages that are used for a remote command when it is called 100 times with an event associated with this command is shown below

MoteID	1 hop	2 hops	Mixed
ID0	218	221	425
ID1	222	214	451
ID2		218	236
Total(messages)	440	653	1112
Time(secs)	3.38	4.56	9.86

Fig. 26 – RMI Operation performance

The observations that can be drawn from the above table ate that as the number of hops are increasing and more components using the RMI layer are increasing there is more message loss and more number of messages are used.

CHAPTER 13 – CONCLUSIONS

13.1 Summary

The project framework enables the user to design a global application in Cadena tool and to specify the commands the user want to issue to the network to observe interesting data from the network. The application design can use the generic services provided by the project framework using the virtual node abstraction.

The global application is then converted to local scenarios and the code for these local scenarios are automatically generated by a java tool which can either tested in Tossim simulation framework or can be deployed on real motes.

Finally a java based Graphical User Interface tool is provided to the user so that it can be connected to either Simulation framework or to the actual deployed network to observe the data in the network, debug and monitor the network.

Thus in this way high level abstractions are used, so that the user can only deal with the application logic rather than the underlying low level communication details of wireless sensor networks.

13.2 Future Work

The java tool developed for the generation of the node level scenarios works only for the scenarios when there are no nested components in the model. This tool needed to be modified so that nested component models can also be deployed**.**

The GUI is needed to be more sophisticated to include some provide some graphs and charts display to the user if there are any periodic outputs for the commands.

References

1. TinyOS Tutorial
   http://www.tinyos.net/tinyos-1.x/doc/tutorial/index.html

2. Tython manual
   http://www.tinyos.net/tinyos-1.x/doc/tython/manual.html

3. Pietro Ciciriello, Luca Mottola and Gian Pietro Picco: Building virtual sensors and actuators over logical neighborhoods, Proceedings of the International Workshop on Middleware for Sensor Networks, 2006

4. Cadena 2.0: Manual
   http://cadena.projects.cis.ksu.edu/update/web/cadena-manual.pdf

5. Remote Method Interface
   Tannenbaum. Modern Operating Systems, Second Edition, Prentice Hall.
   Chapter 2: Communication in Distributed Systems