Appendix A
The Atarraya Simulator

A.1 Introduction

The main idea behind the creation of Atarraya – which means fishnet in Spanish – was to test the topology construction protocol named A3 that was being developed as part of this research. The software, as originally conceptualized, was very simple, rigid and tightly coupled with the A3 protocol. However, due to the fact that it was necessary to compare the performance of A3 against other known topology construction mechanisms, the design of the tool was not adequate. Therefore, the decision to build a more generic simulator, in which other topology construction algorithms could be plugged in, and have a single platform where to evaluate them all under the same conditions, was necessary. Then, the concept of topology control was also expanded to include topology maintenance algorithms, and several of these mechanisms were designed and included as well.

The final result is Atarraya: a generic, Java-based, event-driven simulator for topology control algorithms in wireless sensor networks. As with any simulation tool born out of a research effort, Atarraya is still in development; however, in its current state, it is an excellent tool not only for research, to develop and test new topology control algorithms, but also for teaching. Atarraya’s graphical user interface shows how topology control protocols work, and how they shape topologies during their execution. In addition, Atarraya includes necessary mechanisms to experiment with classic theoretical results related to topology control in wireless sensor networks, such as the giant component experiment, calculation of the critical transmission range (CTR), calculation of the Minimum Spanning Tree of a graph, and others.

In this appendix the basics of Atarraya are presented along with its internal structure, so the reader knows how to develop and plug in new topology control algorithms and protocols, and a brief guide on how to use the tool. All explanations and descriptions included in this document are related to Atarraya’s version 1.0, which is the version that was used to run all the experiments included in the book. Future versions and new features will be documented on the project’s Website at http://www.csee.usf.edu/~labrador/Atarraya.
A.2 Description of Atarraya’s Internal Structure

In this section, the internal structure of Atarraya is described. First, its main functional components and Atarraya’s class tree are presented. Then, the structure of the protocols is described in more detail, including how they communicate with the main class and with other protocols, how to initialize the nodes, and how to handle protocol events.

A.2.1 Abstract Design and Functional Components

This section describes the main functional components of the simulator and how they interact among themselves. The functional components offer the “big picture” necessary to understand the critical components of Atarraya. Figure A.1 presents a global view of the internal structure of the simulator, which consists of the main simulator thread, the node handler, and the batch executor. The elements of this structure are described in this section.

A.2.1.1 The Main Simulator Thread – The the_sim Class

This is the core of the system. The simulator thread, defined in the class the_sim, is in charge of fetching the next event from the simulation event queue, and sending the event to the node handler for execution. An instance of this class is created by the
method StartSimulation() whenever a simulation is executed. This class contains the event queue, the simulation clock, the display manager, the database with the data about the nodes, and the simulation agent, which is in charge of storing the simulation results for the reports in the respective logs.

When an instance of the the_sim class is created, it is necessary to add the initial events to the queue before the thread is started. The first thing the thread will do once started is to check if there are any events in the Event Queue. If the thread is started without any events, it will consider that an error has occurred, and the simulation will be suspended.

Once the first events have been loaded into the queue, the simulator thread can get started. The thread starts a loop that will execute until one of the three termination conditions is true: there are no events in the queue, all the nodes have reached the final state in the topology control protocol, or the protocols have called for the end of the simulation (for example, the topology maintenance protocol has found that the sink has no more neighbors, so the network is dead). If the first condition occurs and the simulator has not been notified that the protocols finished execution, it means that there was an error during the simulation, and it will be notified on the simulation report.

In the loop, the first thing the thread does is to verify if the event is valid. If so, the event will be registered (if this option was selected by the user), the simulation clock will be updated, and the event will be sent to the NodeHandler. There, the event will be delivered to the appropriate EventHandler according to the respective protocol the event belongs to. Once the event is executed, the simulator will go back to the loop and start again the process. The simulator updates the clock with the execution time of the events based on the fact that all the events in the queue are sorted by their projected execution time, so there is no such thing like a trip to the past.

Once the simulator breaks the loop by any of the finalization conditions mentioned above, the thread goes to the report construction section, saving all the events and statistics, as selected by the user. This section also takes into account whether or not the simulation is part of a batch execution, in which case all the data from all previous executions is kept until the last one finishes. All this information is stored in data structures that are stored in the report files after the simulation is finished. Reading this section of the code will provide the user with information about all the options for each configuration of report in both single and batch simulation cases. Once the simulation and the report building section finish, the thread ends too.

In the current version of the simulator, just one simulator thread can run at a time because there is only one data structure to store the topology, which is localized in the atarraya_frame class. Individual instances of the data structure running several simulations in parallel will consume all the resources of the Java virtual machine, especially if the network topologies are big.

A.2.1.2 The Protocol Manager – The NodeHandler Class

This class is in charge of defining the protocols to be used in the simulation and routing the event to the appropriate protocol once received from the simulation thread.
The NodeHandler class defines the four possible protocols that a node can have running during a simulation: Topology construction, topology maintenance, sensor-data management, and communication-routing protocols. Given that there are different algorithms for each type of protocol, the main purpose of this class is to make that selection transparent to the rest of the simulator, so that no detail about the selection is required in order to execute the simulation. When a simulation is started, this class creates the instances of the selected protocols in each of the four different categories. The simulation thread sends the next event from the event queue to the NodeHandler class. Once the event is received, it is routed to the appropriate protocol based on the protocol identifier included in the event.

### A.2.1.3 The Multiple Operation Thread – The BatchExecutor Class

The main purpose of the BatchExecutor thread is to perform operations that require multiple executions, such as creating a set of topologies, performing a large number of simulations, and the Giant Component test. Since these operations are run on a thread independent from the main one, the graphical user interface does not freeze while these operations are being executed, which allows for the interaction between the user and the simulator even while some of these operations are running in the background. This class is instantiated whenever one of the mentioned operations is started.

### A.2.1.4 The Display Manager – The newpanel Class

The display manager, or newpanel class, is the one in charge of the graphical representation of the topologies. The heart of this class is the override of the Paint method of this class that extends a Panel class. All the painting options for the topology are defined in this method. The other methods perform minor but necessary actions like obtaining information about the options, providing coordinates from the deployment area, etc. This class was defined as a private class of the atarraya_frame class so it can have direct access to the topology data structure.

Atarraya provides several options for topology visualization, which can be seen in more detail in the visualization options in Figure A.12. The most relevant visualization options are the following:

- **MaxPower Topology:** This is the original view of the topology with all nodes transmitting at full power, and all the links that their unit disks provide.
- **Single selected network configuration or tree:** In this view the user defines which of the Virtual Network Infrastructures (VNI) he or she wants to see. More on VNI later in Section A.4.3. The default configuration is Black in most protocols.
- **All network configurations:** If several configurations are defined in a certain topology, this view allows the user to see all of them and appreciate the differences between them.
- **Active network configuration:** Each node is assumed to be able to maintain several VNIs, but use only one at a time. This view allows the user to see in real-time in which network configurations the nodes of the topology are.
A.2 Description of Atarraya’s Internal Structure

A.2.2 Atarraya’s Class Tree

In this section the class tree of Atarraya is described and a brief explanation of the structure and mission of the current internal structure of the application is provided. The classes in Atarraya are organized in three packages:

- **Atarraya**: the main functional elements are stored here, such as the main frame, the simulation agent, and the display manager.
- **Atarraya.element**: This package contains the classes that model the data structures, like the node, VNI, routing table, etc.
- **Atarraya.event**: This package contains the classes related to the protocols and the definition of the event queue.

A.2.2.1 The Atarraya Package

The Atarraya package is the main package of the simulator. It contains the following classes:

- **Main** class: This is the launcher of the simulator. It invokes the title frame and the main frame.
- **atarraya_frame** class: This is the main class of the simulator. In this class we find the definition of the graphical user interface and the simulator core.
- **newpanel** private class: This class defines the operation related to the visualization panel for the topologies: painting, selection of coordinates, selection of nodes, grid, etc. It is a private class of the atarraya_frame class so the newpanel class has direct access to the data structures.
- **the_sim** private class: This class defines the structure of the thread that simulates a scenario; in other words, this class is the simulation executor. It was made private also to preserve the direct access to the data structures.
- **BatchExecutor** class: This class is in charge of executing operations that involve multiple scenarios, being that create multiple topologies, or simulate multiple scenarios. The advantage of using a separate class is that it creates a different thread that freezes the main frame while executing.
- **constants** interface: This interface defines the standard values for multiple variables. Given that many classes must share a set of standard values, the use of the constant interface allows this without having to define identical variables on each class.
- **FrameLogo** class: Initial frame with the logo of the simulator.
- **AboutFrame** class: Frame that contains the “About us” message.

A.2.2.2 The Atarraya.element Package

This package contains the classes that define the elements that will be used in the simulator for storing information. These classes are:

- **node** class: This class represents all the information about a single node and all the operations that can be performed on the individual nodes.
- **register** class: This class contains the information that the simulator has about the neighbors of a node. There is a difference between the information that the
A.2.2.3 The Atarraya.event Package

This package contains the classes that define the structure of the events, the event queue, and the event handlers that will execute the events accordingly. The main classes are:
• *event_sim* class: This class defines the information of an event: source, destination, type of event, embedded information, configuration, layer that generated the event, etc. This class contains all the information that the event handler requires to execute the event properly.

• *eventQueue* class: This class defines the queue of events that Atarraya uses during the execution of a simulation. Events are added to the queue in an organized way, based on the projected execution time of the event, so the event on the head of the queue is always the closest to the present time.

• *EventHandler* class: This is a family of classes that define the protocols. Each class has to define some initialization operations for the nodes, how the events will affect the state of the nodes, the data structures, other events that get triggered as a consequence of the occurrence of one event, etc. A class of this type needs to be designed if a new topology control algorithm is to be included in the simulator, or if an existing algorithm is to be modified. In its current version, Atarraya supports four types of protocols: Topology Construction, Topology Maintenance, Sensor-Data Management, and Communication-routing protocols. The way these protocols communicate is by generating events of each other’s type.

• *NodeHandler* class: This class defines a data structure that holds the selected options for the four types of protocols. Any new protocol added to Atarraya must be included in the existing list that this class contains.

### A.3.1 Simulation Events

Given that Atarraya is an event-driven simulator, everything that happens during a simulation is an event, so protocols must be defined in terms of cause-effect when certain event occurs. Each of these types of events triggers some internal actions in the node that might modify its status, data structures, etc., and could also cause the generation of new events in the future. The most common examples of events in a protocol are:

#### A.3.1.1 Sending Messages

When a node intends to send a message, it may be addressed to a specific node (unicast) or it may be intended for every neighbor within range (broadcast). Regardless,
the method that a node needs to call on in order to send a message is broadcast. The parameters are the time at which the packet is received, the current time, the id of the sender node, the id of the receiver node (if it is a unicast message), the type of message, the payload, and the VNI corresponding to this package. In general, a message of this kind is assumed to be of the same type as that of the protocol that contains it, which explains why there is no specification of the protocol’s type. More data can be included in this message, like the first sender of the message or source, and the final destination of the packet, in case it is a message that will travel through multiple hops.

    broadcast(temp_clock+getRandom(MAX_TX_DELAY_RANDOM),
    temp_clock, sender, -1, HELLO, temp_data,temp_vni);

The method broadcast is in charge of generating the reception events in the neighbors within communication range of the sender node, if the recipients and the sender node are active. The method broadcast is defined on the atarraya_frame class.

A.3.1.2 Receiving Messages

When a node receives a message, it calls the event determined by the type of message. If the message is supposed to be a unicast transmission, the node verifies if the id of the destination node matches its own. If that is the case it continues with the execution of the algorithm. If the receiver and the destination do not match, the node ignores the packet. Now, if the message was designed to be a broadcast transmission or the node needs to snoop in the packets not addressed to itself, it can ignore the receiver destination verification and just continue with the execution of the protocol.

    case HELLO:
    if(receiver == destination || destination == -1){
        ... //If the message was a unicast and the receiver was the
        destination or the packet was a broadcast
    }else{
        ... //If the packet was addressed to another node and this
        is snooping
    }
    break;

A.3.1.3 Programming a Timeout

Sometimes a node needs to wait some time in order to perform a certain action. The definition of a timer is crucial for this type of operation. A timer in Atarraya is an event that a node programs addressed to itself in the future. The parameters are very similar to the ones provided to the broadcast method, with the difference that here the parameters of a new event, that will be included directly in the simulation queue, are also specified. Since this is a reflective event, the sender and the receiver have the same value. Also, in the declaration it is necessary to specify the target protocol of this event in the variable type. In the example, the node sender is programming
itself an event of the type \textit{PARENT\_RECOG\_TIME\_OUT}, that will be executed in \textit{TIMEOUT\_DELAY} time units.

\begin{verbatim}
  pushEvent(new event_sim(temp_clock+TIMEOUT_DELAY, temp_clock, sender, sender, PARENT\_RECOG\_TIME\_OUT, ",", temp_vni, type));
\end{verbatim}

\subsection*{A.3.1.4 Invalidating a Programmed Event}

A node programs events in the future without knowing what will really happen between the current time and the future event. For example, a node can be programmed to send a message in the future, but for some reason it may also be put to sleep before it can send the message. Since that particular event will not occur, it has to be taken out from the simulation queue, where they are waiting to be executed. Atarraya provides the methods \textit{InvalidateAllEvents} in order to guarantee that a node can cancel events that should not happen. In the example, the sender node eliminates all the events of the current protocol, referent to the VNI \textit{temp\_tree} from the current time forward.

\begin{verbatim}
  InvalidateAllEventsFromIDFromTimeTOfTypeTy(sender, temp_clock, type, temp_vni);
\end{verbatim}

\subsection*{A.3.2 State Labels}

In general, a good number of topology construction protocols use node states to represent the evolution of the protocol. In Atarraya, nodes can be in any of the following four states: Initial, Active, Inactive, and Sleeping states. The definitions of these four states are included in the \textit{Node\_Handler} class. The values defined as the parameters are usually defined in the \textit{constants} interface, and they are all positive integer values.

\begin{verbatim}
  tc_protocol_handler.setLabels(S\_INITIAL\_STATE, S\_ACTIVE, S\_SLEEP, S\_SLEEP);
\end{verbatim}

Given that most topology control protocols implemented in Atarraya are completely distributed, the sink cannot call the end of the protocol because it has no information about the state of all the nodes. That is the reason why Atarraya knows that a protocol has finished when all the nodes have reached the final state. Each topology control protocol can define which states are selected as the final states. This is done in the method \textit{CheckIfDesiredFinalState(int s)} that is defined in every \textit{EventHandler}, which is invoked in the \textit{atarraya\_frame} class when the simulation agent is trying to verify if the topology control algorithm is finished. In the following example, the protocol is selecting the active and the inactive states as the final states of the nodes.

\begin{verbatim}
  public boolean CheckIfDesiredFinalState(int s)
  {
    if(s==active || s==inactive)
      return true;
    return false;
  }
\end{verbatim}

Atarraya stops whenever the nodes of the topology are in any of the selected states, no matter if there are still events in the queue.
A.3.3 Communication with the atarraya_frame Class

Each protocol receives a reference to the instance of Atarraya’s main frame, as defined in the NodeHandler. This reference allows the protocol to have access to variables from the main class. In order to access the variables from the simulator, the protocol needs to use the method father.getVariable(int code), where the parameter code is a defined label for the sets of variables that can be accessed. This list can be found in the constants interface, and in the atarraya_frame class where the getVariable() method is defined. For example, if we need to know how many nodes are in the topology (including the sink nodes), the following line returns this value:

\[
\text{tam} = (\text{int}) \text{father.getVariable(\text{NUMPOINTS})}
\]

When the protocol needs to get information about a node or modify it, the method to use is getNode(int id), where id is the unique id of the node. In order to set node \(i\) in the initial state of the protocol in the VNI _vniID, the following line can be used:

getNode(i).setState(initial, _vniID)

A.3.4 Interaction with Other Protocols

There will always be some level of communication between protocols. For example, inter-protocol communication is needed to avoid situations like one node wanting to send a data message without having a route to the sink. Given that in Atarraya every event in the simulation goes to the same queue, it is necessary to determine to which protocol it must send the event to. Each type of protocol has its own identifier label, which is included in the event definition. This allows the Node_Handler to send the event to the appropriate protocol.

One of the premises of Atarraya is to create modular protocols that can be used in as many combinations as possible with the other protocols. Accordingly, protocols in Atarraya can only generate events in other protocols. For example, once a node reaches the final state of its topology construction algorithm, it can notify the topology maintenance protocol to start the maintenance procedure. Most of the times these inter-protocol events are meant to initiate or stop certain activity, so the protocol and how it works internally are completely independent, but the other protocols can decide the starting points. The following example illustrates a topology construction protocol when it invokes the topology maintenance protocol.

\[
\text{pushEvent(new event_sim(temp\_clock+DELTA\_TIME, temp\_clock, receiver, receiver, INIT\_EVENT, "",temp\_tree,TM\_PROTOCOL))};
\]

A.3.5 Initialization of Nodes and the Initial Events – The init_nodes and the initial_event Methods

The init_nodes(int vni) method is used to set the nodes ready to start the execution of the simulation. Nodes are set to their initial states, and any previously defined events
regarding other protocols and all necessary variables are set to their default values. This method is invoked in the `StartSimulation` method in the `atarraya_frame` class, for all nodes, including the sink. The following code is an example of a `init_nodes` routine, in which every node is set to its initial state, every state label is defined, any existent programmed event in the queue is cancelled, and the execution of the topology maintenance and sensor and data management protocols are reset.

```java
public void init_nodes(int _vniID){
    tam = (int)father.getVariable(NUMPOINTS);
    _clock = father.getVariable(CLOCK);
    temp = 0;

    for(i=0;i<tam;i++){
        getNode(i).setState(initial,_vniID);
        getNode(i).SetInfrastructureStarted(_vniID,true);
        getNode(i).defineLabels(initial,active, inactive, sleeping);

        if(TM_Selected){
            //stop all future event from the TM protocol
            pushEvent(new event_sim(_clock+getRandom(PROCESSING_DELAY),
                                     _clock, i, i, RESET_TM_PROTOCOL, "",_vniID,TM_PROTOCOL));
        }

        if(SD_Selected){
            //stop all future event from the sensor-data protocol
            pushEvent(new event_sim(_clock+getRandom(PROCESSING_DELAY),
                                     _clock, i, i, RESET_QUERY_SENSOR, "",_vniID,SENSOR_PROTOCOL));
        }
    }
}
```

The `initial_event(int _id, int _vniID)` method is used to define the first events to be included in the queue, before the simulation agent is started. Remember that if the simulation agent finds an empty queue it considers that the simulation has finished in an incorrect way. This method needs two parameters: the ID of the node that will perform the first event, and the VNI ID. This method is also invoked in the `StartSimulation()` method in the `atarraya_frame` class, but only for the sink nodes. If no sink nodes are defined in your topology, make sure that events are included in the queue using the `init_nodes(_vniID)` method.

### A.3.6 The HandleEvent Method

This method is the core of the protocol, as it defines the actions taken by the protocol when an event occurs. The unique parameter that this method receives is the event taken from the event queue.

The events are classified based on an event label. Each protocol defines a set of labels for all the events that it uses. These labels are defined in the `constants` interface. The first action taken by the `HandleEvent` method is to recover all the
fields from the event and store them in temporary variables. Depending on the nature of the protocol, the classification of the events can be done in different ways: Label-then-State or State-then-Label. In the first case, the most important information is the label of the event, which becomes the main classification factor. Once the label is found, the code inside determines if the state of the node is important or not for the execution of the actions associated with the event. In the second case, the most important information is the state of the node. This methodology is useful when there are not many types of events but each type is interpreted differently based on the state of the node.

The following code shows the HandleEvent method from the example protocol presented in this section, the SimpleTree.

```java
public void HandleEvent(event_sim e){
    int code = e.getCode();
    int sender = e.getSender();
    int source = e.getSource();
    int final_destination = e.getFinalDestination();
    int receiver = e.getReceiver();
    int destination = e.getDestination();
    double temp_clock = e.getTime();
    String temp_data = e.getData();
    int temp_vni = e.getTree();

    switch(code){
    case INIT_NODE:
        init_node(temp_vni, receiver);
        break;

    case INIT_EVENT:
        initial_event(receiver,temp_vni);
        break;

    /*
    * This event is when a node will start
    * sending Hello message to its neighbors
    */
    case SEND_HELLO:

        //The node will clean the candidates from
        //the neighbor’s hellos messages
        getNode(sender).cleanCandidates();

        getNode(sender).setState(S_IN_SEARCH,temp_vni);

        //Sending Hello message to all neighbors
        broadcast(temp_clock+getRandom(MAX_TX_DELAY_RANDOM),
                temp_clock, sender,-1, RECEIVE_HELLO, temp_data,temp_vni);

        //Final timeout for evaluating candidates and adopt
        //children nodes
```
pushEvent(new event_float(temp_clock+TIMEOUT_DELAY, temp_clock, sender, sender, LISTEN_4_REPLY_TIME_OUT, "",temp_vni,type));

break;

} //This event is when a node received a Hello message from its parent
/*
case RECEIVE_HELLO:
   //If the node does not have a parent
   //in the current tree...
   if(!getNode(receiver).isCovered(temp_vni)){
      //Decompress the data in the message
      temp_data_array = temp_data.split("@");
      try{
         //The level of the parent is coming in the data
         temp_data_int = Integer.parseInt(temp_data_array[0]);

         //The sink address of this VNI
         temp_data_int2 = Integer.parseInt(temp_data_array[1]);
      }catch(Exception ex){ex.printStackTrace();}
      //Change the state of the node from the initial state
      getNode(receiver).setState(S_VISITED,temp_vni);

      //Define the parent in the current tree and
      //the sink address
      getNode(receiver).setParent(temp_vni,sender,sender,
          temp_data_int2);

      // Define the level
      getNode(receiver).setLevel(temp_data_int+1);

      // Schedules the Broadcast of the Reply message
      pushEvent(new event_float(temp_clock +getRandom(PROCESSING_DELAY), temp_clock,receiver, receiver,
          SEND_REPLY, "",temp_vni,type));

      //If it is not a parent after TIMEOUT_NO_PARENT units,
      // it will go into S_SLEEPING mode
      pushEvent(new event_float(temp_clock+TIMEOUT_NO_PARENT, temp_clock, receiver, receiver, END_TIMEOUT_NO_PARENT,
          "",temp_vni,type));
   }
}
break;

} //This event is when the timeout for listening to other
* neighbors finishes, and the node can send its own reply
* to the parent.
*/
case SEND_REPLY:

    //Parent’s ID
    temp_ID = getNode(sender).getParentID(temp_vni);

    //The metric can be whatever the user needs it to be:
    //energy, ID, etc.
    metric = getNode(sender).getMetric(temp_ID);

    //You can use thresholds to limit responses...
    //For example, if energy is low, do not answer
    //y=java.lang.Math.random();
    y=0;
    if(metric > y){
        //Determine the metric to send
        temp_data = "" + metric;

        //Broadcast the message
        broadcast(temp_clock+getRandom(MAX_TX_DELAY_RANDOM),
                    temp_clock, receiver, temp_ID, RECEIVE_REPLY,
                    temp_data,temp_vni);
    }
    break;

    /*
    * This event is when a node received a reply
    * from a Hello message from its children or neighbors.
    */
case RECEIVE_REPLY:

    //If the parent is the one that receives this message
    if(receiver == destination){

        //Obtaining the neighbor identifier in the node
        temp_ID = getNode(receiver).getNeighborIndex(sender);

        //Obtaining the metric that the node sent
        try{
            metric = Double.parseDouble(temp_data);
        }catch(Exception ex){ex.printStackTrace();}

        //Include a new neighbor from which the node heard
        //a hello message
        getNode(receiver).addCandidate(new candidate(temp_ID,
                                                 sender,metric));
    }
    break;
case LISTEN_4_REPLY_TIME_OUT:
    i=0;
    sw=true;
    temp=0;
    temp_data="";

    //Number of candidates
    temp2=getNode(sender).getNumCandidates();

    if(temp2>0){
        //Changing the state of the node to be parent
        getNode(sender).setState(S_PARENT,temp_vni);
        getNode(sender).setParent(temp_vni, true);
    }
    //Initiate the TM protocol only on the active
    //nodes of the topology!!
    if(TM_Selected){
        pushEvent(new event_sim(temp_clock+DELTA_TIME,
            temp_clock, receiver, receiver, INIT_EVENT,
            "",temp_vni,TM_PROTOCOL));
    }
    //Initiate the sensor querying protocol only
    //on the active nodes of the topology!!
    if(SD_Selected){
        pushEvent(new event_sim(temp_clock+DELTA_TIME,
            temp_clock, receiver, receiver, INIT_EVENT,
            "",temp_vni,SENSOR_PROTOCOL));
    }
    //Initiate the COMM protocol only on the active
    //nodes of the topology!!
    if(COMM_Selected){
        pushEvent(new event_sim(temp_clock+DELTA_TIME,
            temp_clock, receiver, receiver, INIT_EVENT,
            "",temp_vni,COMM_PROTOCOL));
    }

    for(i=0;i<temp2;i++){
        //Get the i th candidate
        temp_cand = getNode(sender).getCandidate(i);

        //If the ID of the candidate is greater than 20,
        //then it will be a child!!
        if(temp_cand.getID()>20){
            //Adding the new child
            getNode(sender).addChild(getNode(sender).
                getNeighbor(temp_cand.getIndex()));
        }
    }

    //Initiate the TM protocol only on the active
    //nodes of the topology!!
    if(TM_Selected){
        pushEvent(new event_sim(temp_clock+DELTA_TIME,
            temp_clock, receiver, receiver, INIT_EVENT,
            "",temp_vni,TM_PROTOCOL));
    }
    //Initiate the sensor querying protocol only
    //on the active nodes of the topology!!
    if(SD_Selected){
        pushEvent(new event_sim(temp_clock+DELTA_TIME,
            temp_clock, receiver, receiver, INIT_EVENT,
            "",temp_vni,SENSOR_PROTOCOL));
    }
    //Initiate the COMM protocol only on the active
    //nodes of the topology!!
    if(COMM_Selected){
        pushEvent(new event_sim(temp_clock+DELTA_TIME,
            temp_clock, receiver, receiver, INIT_EVENT,
            "",temp_vni,COMM_PROTOCOL));
    }

    for(i=0;i<temp2;i++){
        //Get the i th candidate
        temp_cand = getNode(sender).getCandidate(i);

        //If the ID of the candidate is greater than 20,
        //then it will be a child!!
        if(temp_cand.getID()>20){
            //Adding the new child
            getNode(sender).addChild(getNode(sender).
                getNeighbor(temp_cand.getIndex()));
        }
    }
// Sending the packet
broadcast(temp_clock+getRandom(MAX_TX_DELAY_RANDOM),
temp_clock, sender, temp_cand.getID(),
ACCEPTANCE_MESSAGE, temp_data,temp_vni);

break;

/*
 * This event is when a node is accepted by the sender
 * node and it gets permission to start its own branch
 * of the tree.
 */
case ACCEPTANCE_MESSAGE:
    temp=0;
    if(sender==getNode(receiver).getParentID(temp_vni)&&
       receiver == destination){

        pushEvent(new event_float(temp_clock
            +getRandom(MAX_TX_DELAY_RANDOM),
            temp_clock, receiver, -1, SEND_HELLO,
            ""+getNode(receiver).getLevel() +"@
            +getNode(receiver).getSinkAddress(temp_vni),
            temp_vni,type));

    } break;

/*
 * This event is when a node did not received
 * an ACCEPTANCE_MESSAGE.
 */
case END_TIMEOUT_NO_PARENT:
    if(!getNode(sender).getState(temp_vni) == S_VISITED){
        frame_repaint();
        getNode(sender).setState(S_SLEEPING,temp_vni);
    }

    break;
}
A.3.7 SimpleTree: An Example of a Topology Construction Protocol

In order to illustrate all these concepts, a simple example of a topology control algorithm follows. The selected example is a hierarchical topology construction protocol based on the growing a tree technique (see Section 8.2.1). The idea is to illustrate some common message exchange sequences, the use of timeouts, and how the modification of the status of the node modifies the execution of the protocol. The protocol works based on the following assumptions:

- The growing a tree protocol leaves every node active, except those with ID number less than 20.
- The nodes have no information about their position and have no list of neighbors.
- Every node starts in a unvisited state.
- The sink is the initiator of the process.
- The protocol ends when every node is in active mode or in sleeping mode.
- The initial topology is connected.

The protocol, step by step, works as follows:

- The sink node initiates the protocol sending a Hello message. It includes its address and level (number of hops from the sink), which in the case of the sink it is equal to 0. The sender node programs a timeout to stop listening for answers from its neighbors.
- All unvisited nodes on the transmission range of the sender node that received the Hello message answer back with a Reply message, set the sender as their default gateway, and change their status to In-Process mode. The receiver nodes at the same time set a timeout in case they do not receive an acknowledgment from their default gateway (the sender node).
- Once the timeout of the sender expires, this node checks the list of neighbors that answered back with the Reply message. If the sender node did not receive any answer back, it turns itself off and changes its status into a Sleeping mode. If the sender node received at least one answer back, it goes into Active mode and sends an unicast message to each one of them in order to let them know they were selected. All neighbors except those with ID number less than 20 will receive this unicast message.
- Once a node receives the confirmation message from its default gateway, it waits a random amount of time and sends its own Hello message to discover new unvisited nodes.
- If the timeout of a node in In-Process mode expires, it means that the node was not selected, so it turns itself off and goes into Sleeping mode.
- Once the node finishes its process and ends up in an Active mode, it starts the topology maintenance, sensor and data management, and communication protocols.

Although describing a protocol in words is useful for understanding its operation, they are more rigorously defined by Finite State Machines, especially in those
cases where nodes change several times from one state to another during the execution of the protocol, and by a timeline of message exchanges. For this example protocol, these diagrams are shown in Figure A.2.

The first part of the algorithm is a HELLO-REPLY sequence that is used by many protocols in the neighborhood discovery process. The message sequence is simple: one message announces the presence of a node, and a set of nodes, whose size is unknown, answer back with a reply message. Given that the initiator has no idea of how many nodes are within its transmission range, it waits for a certain amount of
time. This timer can be static (a fixed value or a random value defined on the fly) or dynamic (value is changed after the reception of a new reply message).

The second part of the protocol consists of the selection and notification processes. In this case, the sender node selects the next generation of possible active nodes based on a policy (all nodes with ID less than 20), and notifies them one by one using unicast messages.

The third part is the initiation of the other protocols. Remember that topology construction protocols are only used to reduce the size of the initial topology, but they do not necessarily take care of maintaining this topology during its operation, or send data messages to the sink. If the user is at the early stages of the protocol design, running experiments with the topology construction protocol only is very convenient; however, for more complex experiments that include network lifetime measurements it is necessary to start the topology maintenance protocols.

### A.4 How to Use Atarraya

Atarraya offers a variety of options for designing and experimenting with topology control algorithms. This section provides an in-depth user guide on how to use the simulation tool and all its available options.

The first step is to have a clear understanding of the simulation scenarios to be run. Here, the user needs to know in advance which protocols he or she wants to use; whether the experiment is just a preliminary test or an exhaustive performance evaluation; what is the nature of the topologies that she is planning to use; what type of statistics are needed, and so forth. In this section, these questions are answered in order for a user to create and run successful simulations with Atarraya.

#### A.4.1 Selection of the Protocols

Atarraya includes four types of protocols: Topology construction, topology maintenance, sensor-data management, and communication-routing protocols. Atarraya can be set to work in either of the following two modes related to topology control: Topology construction only, or All protocols. The first mode is designed to test a specific topology construction algorithm and measure the initial reduced topology that the algorithm produces. In order to select this mode, the user only needs to select a topology construction protocol.

The second mode is intended to test not only the reduced topology but the lifetime of the network, based on the combination of all the protocols, i.e., topology construction and topology maintenance. In order to select the second mode the user needs to select a protocol in each of the protocol categories, i.e., select a topology construction and a topology maintenance protocol, otherwise Atarraya will not allow the user to run the experiment.

The topology construction protocols included in Atarraya are based on algorithms presented in published papers. Currently, Atarraya includes the algorithms evaluated in this book, i.e., A3, EECDS, and CDS-Rule-K. Although all types of
topology construction protocols might be implemented, such as those based on changing the transmission range of the nodes, hierarchical protocols, cluster-based protocols, etc., the current version of Atarraya includes only those based on the Connected Dominating Set concept. In addition, and for the sake of comparison with a wireless sensor network without topology control, Atarraya offers a protocol that does not reduce the topology at all, called JustTree. The only service that this protocol provides is the creation of a forwarding tree to implement the constant gateway forwarding protocol.

Atarraya also includes all the topology maintenance techniques included in Part II of this book. They are generic algorithms that work with any of the topology construction protocols. The simplest topology maintenance protocol included in Atarraya is the No Topology Maintenance protocol, which does nothing to maintain the initial topology, but monitors it until it dies. The protocol is in charge of informing Atarraya when this event happens. In Atarraya, the termination policy is defined as the moment at which the sink stops receiving information from the nodes.

The sensor-data management protocol models the behavior of the sensors, regarding variables like data transmission frequency, data aggregation policies, etc. Atarraya provides a simple protocol for sending and receiving messages without data aggregation. Nodes transmit data packets at predefined times, and forward every received data packet based on the forwarding policy explained in the following paragraph.

Although communication or routing protocols are not the focus of this simulator, some kind of routing procedure is necessary so packets can reach the sink. Given that the topology control protocols implemented in Atarraya are designed to produce a tree-like reduced topology, the tool provides a very simple forwarding algorithm that allows packets to reach the sink node: The constant gateway forwarding protocol. In this protocol, packets are always forwarded to a default gateway unless the destination of the packet is a direct neighbor, in which case it will be sent directly to that node. In a tree-like topology, the gateway of a node is simply its parent node. In this fashion, the packet will finally reach the sink node, since it is the root of the tree.

Atarraya does not include any routing protocol, but defines a data structure that models a routing table. This data structure allows users to develop more advanced routing protocols than the one currently implemented. In addition, the routing table can store a limited amount of packet sequence numbers (events have a field for this purpose) that allow the implementation of other forwarding algorithms like flooding-based protocols, or save the last versions of the routing protocol information packets, like routing tables on a Distance Vector protocol, or the last neighborhood information in a Link-State protocol.

In Atarraya’s graphical user interface, the panel named Atarraya presents the available protocols. In addition, this panel contains the controls for the simulation agent, the report configuration options, simulation events and statistics panels, and the batch simulations controls. This panel is shown in Figure A.3.
A.4 How to Use Atarraya

Fig. A.3 Simulation control panel.

A.4.1.1 Other Protocols

In order to write this book, several experiments were run to validate analytically derived equations to show special effects or behaviors of topology control algorithms. This section describes five tools included in Atarraya that are very well suited for educational purposes, as they allow users to:

- Calculate the Critical Transmission Range (CTR) based on the formulas of Penrose–Santi and Santi–Blough.
- Reproduce the experiment to obtain the Giant Component figures: Greatest Connected Component and Ratio of Connected Topologies.
- Reproduce the experiment that proves connectivity of the CTR formula of Santi-Blough for 2 dimensional deployments.
- Calculate the Minimal Spanning Tree on a given graph and provide the sum of the selected edges.
- Save the neighborhoods of a graph in a file.

The first three tools were utilized in Chapter 7. The user can reproduce those experiments with different parameters if so desired. The Minimal Spanning Tree of a graph is still a classical tool for graph analysis – Prim’s algorithm was implemented. Regarding the last tool, the information provided by it can be used to define and solve linear programming optimization problems on graphs, like finding a minimal set cover of the graph. The panel that contains all these tools is shown in Figure A.4.
A.4.1.2 Energy and Communications Model

In the design of Atarraya simplicity and focus on reaching a better understanding of the behavior of the topology control algorithms was embraced. As such, several assumptions were made to make the simulator simpler while still good enough for achieving its main objective. These assumptions are related to the energy and the communication models utilized in the tool.

The energy model included in Atarraya is based on the following formulas, taken from [47]:

$$E_{TXbit} = E_{elect} + (E_{amp} \cdot (\pi r^2))$$

$$E_{RXbit} = E_{elect}$$

In addition, Atarraya also makes the following assumptions:

- During the idle time, a node does not spend energy. Even though this assumption has been proven untrue because being idle might be as costly as receiving data, this is still an assumption that can be done in most experiments, since the most important factor is the overhead in terms of message exchange and its associated cost.
- The nodes are assumed to have one radio for general messages and a second radio for control messages: The main radio is used in all operations when the
node is in active mode, and the second one (low power cheaper radio) is used
to send and receive control packets to “wake up” the main radio. Only the main
radio can be turned off, which means no messages will be received and no energy
will be used. The secondary radio is assumed to use half the energy of the main
radio.

- The sink node has an infinite source of energy. In general, the sink node is assumed
to be powered from an external source of energy.

The communication model used in Atarraya is based on the following assumptions:

- The communication range of the nodes is a perfect symmetric unit disk. If \( d_{x,y} \leq
r_x \rightarrow x \) and \( y \) can see each other.

- A constant bit error rate (BER) is defined for the whole network. This is a simple
implementation of an error model. Whenever a packet is going to be sent, a
random number is generated and compared to the message error rate (that depends
on the size of the message). If the random number is greater, the message
is received, otherwise it is lost. The default value for the BER is 0, which means
there is no packet loss. No sum of partially received packets will build a complete
packet.

- Atarraya assumes that there exists a Data Link Layer that deals with packet losses
and retransmissions, but it does not model this. In order to model some of the
consequences of the operations of the MAC layer, the packets are delayed a random
amount of time in order to model delays occurring due to retransmissions,
contention, etc. The variable that defines the maximum delay value can be found
in the constants interface, by the name of MAX_TX_DELAY_RANDOM. The default value for this variable is 0.2 time units.

**A.4.2 Type of Experiments**

Atarraya offers two types of experiments: single topology based, that uses the visual
representation of the topology, and the batch execution mode that simulates a large
set of topologies. The single topology based type is good for protocol design and
debugging. The batch type is made for full scale evaluation and analysis.

During the protocol design phase, it is very important to have the capability to
run a small number of controlled topologies one at a time to be able to compare
the results of several runs on a known scenario. This is a debugging phase where
many changes are introduced in the protocol until it behaves as intended. During
this process, a visual representation of the topology and the performance of the
topology control algorithm is very helpful.

Once the protocol has reached a stable version that has worked well in several
single topologies, the protocol needs a more exhaustive test with a larger number
of topologies in order to analyze its average behavior. The batch execution mode
allows the researcher to run simulations with hundreds of random topologies. In
this case, the visual representation of the topologies is not necessary; actually, it
would slow down the simulation process.
In the single topology mode, the topology is loaded using the Load Topology item in the File menu tab, or the Deployment Options tab, which is generated by the user (the generation of topologies will be explained in the following section). In the batch execution mode, several topologies are loaded from files selected by the user. The batch execution controls are located in the Atarraya panel, as shown in Figure A.3.

In order to increase the randomization of the simulation process, Atarraya introduces some noise on some common processes in the network, like message transmission delay, so each instance of a simulation, even working on the same topology, would produce different outcomes. Atarraya allows multiple replicas of each topology, which is useful to obtain the average results and the variability of a single topology that shows the confidence of the algorithm.

A.4.3 Structure of a Topology

A topology in Atarraya is composed of four basic elements: Deployment area, regular nodes, sink nodes, and virtual network infrastructures or VNI. The deployment area is an abstract concept, which is useful for visualization purposes. It is a rectangle in which the user deploys the nodes of the network. In order to define the deployment area, the user needs to define its width and height.

The set of regular nodes is usually the biggest set of elements in the topology. They are in charge of monitoring the environmental variable or variable of interest, sending this information to the sink and routing packets. As with any wireless sensor devices, regular nodes are very limited in terms of resources.

The sink nodes are special nodes that, in most scenarios, are included to receive the information from all active nodes in the network. They serve as bridges between the wireless sensor network and any type of external network used to transport the sensed data to its final destination somewhere else in the Internet. In some cases they are also in charge of initiating, executing, and/or controlling the topology construction and maintenance protocols, routing protocols, etc.

Despite the fact that in real life the hardware configuration of the sink nodes is different compared to a regular wireless sensor device, Atarraya uses the same data structure to model both nodes, with the main difference that sink nodes are considered to have an unlimited amount of energy.

One contribution of Atarraya is the introduction of the concept of Virtual Network Infrastructures (VNI). This idea is an abstraction used to allow overlaying topologies over the same set of nodes. Imagine having several sink nodes on the initial topology, and that each one builds a reduced topology rooted at itself. Using this abstraction we can rotate not only sink nodes, but also complete topologies. This approach was used to implement the static global topology maintenance techniques defined in Chapter 11. The initial idea was to let the network operate in the same way the lights in a Christmas tree rotate: a certain number of subsets take turns or shifts to be active during a certain amount of time, and go to sleep until the next turn.
Atarraya identifies the different VNIs with numbers from 0 to 6, and visually with colors: Black, Red, Blue, Green, Orange, Pink, and Yellow. Each node is assumed to have a separated data structure for each VNI, so the information of each one is independent from each other. Regular nodes have no affiliation with a particular VNI, while each sink node is associated with a VNI to which it serves as a sink node. All sink nodes are assumed to be regular nodes by the VNI, that are not associated with them, keeping their characteristic of unlimited energy.

The following list contains all the available options to define the network topology:

- **Sink or No Sink**: Even though most wireless sensor networks contain one or more sink nodes, Atarraya allows the user to define a wireless sensor network without sinks.
- **One or multiple sinks in a single VNI**: Atarraya allows to define a single network with a single sink or multiple sinks. Figure A.5(a) shows this first case, which facilitates the design because there is only one active element in charge of organizing the protocols. Nonetheless, Atarraya also allows for network designs with multiples sinks in a single network, as shown in Figure A.5(b). Including many sinks distributed in the area of deployment can be a good idea to reduce the average path length; however, having multiple sinks can also cause network partitions, especially if their structures are disjoint.
- **One VNI or multiple VNIs**: Having multiple VNIs is the way in which the static global topology maintenance techniques were implemented, where there are several subsets of topologies and one sink.

### A.4.4 Structure of the Nodes

In terms of the nodes, Atarraya can manage homogeneous networks, in which all nodes have the same characteristics, and heterogeneous networks, where there is at least one node that is different from the others.
When creating a topology, the simulator works based on subsets of homogeneous nodes. Each set defines a family of nodes that share the same characteristics. For example, the user can define a set of “weak” nodes with low transmission range and low energy, and a set of “powerful” nodes with high transmission range and higher energy. Atarraya also allows for different random distributions for the location and energy of each group of nodes, that will allow the user to create denser zones in the topology, or zones with nodes that have less energy.

The data structure that models these families of nodes is called Creation Words. These creation words are created based on the values defined on the Deployment Options panel in the simulator (Figure A.6). The user can create as many creation words as desired: these families can model from a single node to the complete set of regular nodes. In the panel there are two list boxes where the creation words are stored until the topology is created: The regular nodes list (top) and the sink creation words (bottom). There are three buttons for adding a new creation word, removing a selected creation word, and clearing both list boxes.

The parameters that can be defined in a homogeneous family of nodes are:

- Number of nodes.
- Communication radius.
- Sensing radius.
- Size of the area of deployment.
- Position distribution.
- Uniform, with center in \((x, y)\) and the limits defined by the deployment area parameters.
- Normal with center in \((x, y)\), requiring mean and standard deviation of the location.
- Grid H-V: cover the deployment area with a grid where nodes are connected of its horizontal and vertical adjacent nodes in the grid. Number of nodes is not a parameter for this option. Distance between the nodes is the communication radius, \(comm_{radius}\).
- Grid H-V-D: cover the deployment area with a grid where nodes are connected of its horizontal, vertical and diagonal adjacent nodes in the grid. The number of nodes is not a parameter for this option either. The distance between the nodes is \(comm_{radius}\sqrt{2}\).
- Constant position at \((x, y)\).
- Center of area, based on the size of the deployment area parameters.
- Manual: press the button and click on the panel to select the position of the node, as many times as defined on the number of nodes of the subset.

- Energy distribution:
  - Constant value.
  - Normal, requiring mean and standard deviation of the energy function.
  - Poisson, requiring lambda of the energy distribution.
  - Uniform, requiring maximum of the energy function.

- Inter-query time: Frequency of querying the sensor for readings. This parameter is valid only when a sensor-data protocol is selected.

  In the case of the sink node set, two extra parameters must be defined:

  - Sink?: select the check box if you want the nodes in the set to be included as sink nodes.
  - VNI Selection: select the VNI identifier associated to the sinks of the set.

  Atarraya also includes some other special purpose variables, as follows:

  - Inter-query time: Time period between reading the sensor and transmitting the data packet to the sink node.
  - Inter-reset time: Time period between the time-triggered topology maintenance protocols.
  - Energy threshold: Energy percentage differential used to invoke energy-triggered topology maintenance protocols. Every time the energy of a node crosses this energy threshold value, since the last invocation of the topology maintenance protocol, the node will invoke the protocol again. For example, if the battery is fully charged and the energy threshold is set to 0.10, the node will invoke the protocol for the first time when 90% of its energy is consumed.

  All these options can be found in the Deployment Options tab, under the tabs named Main Parameters and Other Parameters. The Deployment Options and Deployment’s other options panels are shown in Figures A.6 and A.7.
A.4.5 Simulation Results

Atarraya offers three types of result logs that can be obtained from a set of experiments: General statistics, network lifetime, and the simulation event Logs. The **general statistics log** can register the state of several variables in a periodical manner, or provide just one snapshot of the network at the end of the simulation. The most usually consulted variables are:

- Simulation clock.
- Number of nodes.
- Number of sink nodes.
- Number of active nodes.
- Number of dead nodes.
- Average node degree.
- Average level of nodes (if level is used by the protocol).
- Number of messages sent and received.
- Number of data messages received by the sink.
- Energy on the tree, energy spent, etc.
- Area of communication coverage.
A.4 How to Use Atarraya

Fig. A.8 Example of statistics generated by Atarraya.

These values are stored in a text file where each individual row contains a snapshot of the variables of the network in the moment at which data was collected. If an experiment includes more than one topology, the user can decide if the data is going to be stored individually per execution, or if all the results are going to be summarized in a single file. The usual format in which data is presented is in comma separated values (csv), which is readable by most data analysis programs, like Excel, Matlab, etc. However, if the user does not want the results to be saved in files, the Report panel has a text area that holds the statistics generated by the experiments in csv format. If just one topology is simulated, the user can use the Stats text area in the Atarraya panel, which presents the results in plain text format. Figure A.8 shows one example of the simulation results generated by Atarraya.

The network lifetime log registers the status of the active topology. This log stores information every time the topology maintenance protocol is invoked, or when a node dies. This specific log cannot be stored in individual files per execution; it is stored in a summarized format in csv format.

The information stored in the network lifetime log by a single topology is represented in four rows. The first row registers the moment in which the information of the network was calculated. The second row represents the number of active nodes.
that can still reach the sink node—those that still can provide information to the sink. This value is important because if the sink gets isolated, no matter how many active nodes remain, all of them are useless because the information they produce gets lost. In the case of having more than one VNI, the program works with the active VNI in the nodes. The third row shows the ratio between the number of active nodes that can reach the sink (value found on the second row) and the maximum number of active nodes. This value is the percentage of active nodes that are still alive and connected to the sink. Finally, the fourth row contains the percentage of covered area by the active nodes in the second row. This information can be very useful in order to compare efficiency between protocols, in terms of number of active nodes and real covered area.

Atarraya offers a way to summarize this log by calculating the average number of active nodes for the complete set of executions that are stored in a single file. The idea is to create a single lifetime function that represents the average number of active nodes and wraps all the individual lifetimes. Atarraya creates a discrete time line, in which the size of each cell is based on the parameter “Time step”. Each cell contains the global average of all experiments during that specific amount of time. The Report panel presents the controls for this operation, as shown in Figure A.9.

The simulation event log registers all the events generated by the simulator during the execution of a single topology. The complete set of events allows the user to debug the protocol by seeing the sequence of operations executed. This information is only available in individual files per topology.
Atarraya not only offers statistics from the complete simulation point of view, but from the individual node perspective. It is always very useful to know the status of the nodes at any given point in time to check if the algorithms are working as desired. This information can be found in the Node Stats panel, as shown in Figure A.10. In order to get the information of a node, the user can do it in two ways:
dragging the mouse pointer to the desired node and clicking over it, or typing the id number of the node in the text field and press the button Generate Stats. Once the node is selected, the point in the topology turns orange and increases in size.

Atarraya also allows for the visual representation of the reduced topology. The main window of Atarraya is divided in two environments: The deployment visualization area (left) and the control area (right), as shown in Figure A.11. One of the panels in the control area contains the visualization options: changing view from MaxPower graph to the reduced topology (Atarraya mode); showing the active nodes (Parent mode), communication and sensing coverage areas, and node id’s; drawing a grid over the deployment the area, etc. The combination of some of these options are shown in Figure A.12, where on the left side of the figure the deployment visualization area with the different presentation options can be seen.
A.5 Future of Atarraya

Atarraya is a very useful tool for testing topology control algorithms, but it is still far from reaching its full potential. There is a lot of room for future improvements. For example, transmission range control topology construction protocols, local topology maintenance protocols, data aggregation algorithms, routing and forwarding protocols, 3D scenarios, etc. could be included. More complex and realistic sensing and communication models and data link layer protocols could be incorporated as well.

It is hoped that making Atarraya freely available the research community will contribute to its expansion. Maintaining Atarraya’s Website with the most recent versions of the tool, its upgrades and features, for the benefit of the entire community and the research area is a major commitment. The main Atarraya Website is at http://www.csee.usf.edu/~labrador/Atarraya. Please contact the authors with your own upgrades, suggestions, corrections, etc.
References

References


References

References

111. Shakespeare: http://www.shakespeare-military.com/
Index

A
A3 protocol, 108
Adaptive self-configuring sensor networks topologies (ASCENT) mechanism, 116
Additive increase multiplicative decrease (AIMD) strategy, 52
Aloha protocol, 24
Angular topology control with directional antennas (Di-ATC) protocol, 92
Automatic repeat request (ARQ) protocols, 32

B
Bandwidth–delay product, 32
Berkeley media access control (B-MAC) protocol, 27
Binary sensing model, 19

C
Carrier sense multiple access with collision avoidance (CSMA/CA) protocol, 24
Christmas tree topology maintenance technique, 133
Classification of topology control mechanisms, 68
Clear channel assessment (CCA), 27
Cluster-based routing, 49
Clusterpow, 119
Common power (COMPOW) protocol, 99
Cone-based topology control (CBTC) protocol, 89
Congestion control, 52
Congestion detection, 52
Congestion detection and avoidance (CODA) protocol, 57
Congestion notification, 52
Congestion reaction, 52
Connected dominating set, 106

D
Data centric routing, 45
Delaunay triangulation (DT), 83
Demand assignment MAC protocols, 26
Density-based radio synchronization, 130
Density-based triggering criterion, 127
Directed diffusion protocol, 45
Directed LMST (DLMST), 100
Directed RNG (DRNG), 100
Distance routing effect algorithm for mobility (DREAM) protocol, 47
Distance vector routing protocols, 43
Distance-2 (D2) coloring algorithm, 111
Distributed relative neighbor graph protocol (DistRNG), 91
Dominating set, 106
Dynamic global energy-based topology recreation – DGETRec, 142
Dynamic global time-based topology recreation – DGTTRec, 142
Dynamic global topology maintenance techniques, 142
Dynamic local topology maintenance techniques, 145
Dynamic topology maintenance techniques, 126, 141

207
Efficiency of the ARQ protocols, 33
Energy aware routing, 48
Energy aware routing (EAR) for low energy
ad-hoc sensor networks protocol, 48
Energy dissipation models, 14
Energy efficient connected dominating set (EECDS), 110
Energy threshold, 191
Energy-based triggering criterion, 127
Error control, 52
Error control mechanisms, 31
Error correction, 52
Error detection, 52
Error model, 15
Error probability matrix, 17
Euclidean distance, 82
Euclidean minimal spanning tree, 74
Event handler, 167
Event reliability, 53
Event-to-sink reliable transport (ESRT) protocol, 54
Exposed terminal problem, 22, 64
Extended connected dominating set (ECDS), 113

Face routing, 47
Failure-based triggering criterion, 127
Fault-tolerant local spanning subgraph (FLSS) protocol, 87
Fixed assignment MAC protocols, 23
Flooding, 45
Floor acquisition multiple access (FAMA) protocol, 26
Flow control, 52
Forward error correction, 31
Friis’ free-space propagation model, 12

Gabriel graph (GG), 82
General statistics log, 192
Geographical adaptive fidelity (GAF) algorithm, 87
Geometric random graphs, 66, 74
Giant component, 75
Global topology maintenance techniques, 127
Go-back-N protocol, 32
Gossiping, 45
Greedy perimeter stateless routing (GPSR) protocol, 47
Growing a tree, 107

Hidden terminal problem, 21
Hop-by-hop reliability (HHR) protocol, 53
Hops stretch factor, 66
Hybrid energy-efficient distributed (HEED) clustering, 115
Hybrid global energy-based topology recreation rotation – DGETRecRot, 154
Hybrid global time-based topology recreation – DGTTRecRot, 154
Hybrid topology maintenance techniques, 126, 153

Idle listening, 23
Independent error model, 15
Independent set, 109
Inter-query time, 191
Inter-reset time, 191

K-Neigh protocol, 95
K-neighbor graph, 93
K-NeighLev protocol, 95

Left hand rule, 47
Length stretch factor, 66
Link state routing protocols, 42
Local minimum spanning tree (LMST) protocol, 86
Local topology maintenance techniques, 127
Location-based routing, 46
Log-distance path model, 12
Low energy adaptive clustering hierarchy (LEACH) protocol, 28
Low power listening (LPL), 27
Low-energy adaptive clustering hierarchy (LEACH) protocol, 49, 114

MACA protocol, 25
MACAW protocol, 25
Maximal independent set, 109
Maximum power graph, 67
Maximum power graph (MaxPowerGraph), 63
Metric-based radio synchronization, 130
Minimum active subnet protocol (MASP), 117
Minimum connected dominating set, 106
Minimum dominating set, 106
Minimum power configuration protocol (MPCP), 117

Network lifetime log, 193
Node activation multiple access (NAMA) protocol, 30
Node degree, 63
Node handler, 167
O
On-demand routing protocols, 44
Overhearing, 23

P
Packet reliability, 53
Packet reservation multiple access (PRMA) protocol, 26
Patterned radio synchronization, 130
Power aware multi-access with signaling (PAMAS) protocol, 26
Prim’s algorithm, 107
Proactive routing protocols, 43
Probabilistic sensing model, 19
Pruning-based techniques, 112
Pump slowly, fetch quickly (PSFQ) protocol, 56

R
R&M protocol, 85
Radio synchronization, 129
Random assignment MAC protocols, 23
Random radio synchronization, 130
Random-based triggering criterion, 127
Randomized forwarding, 45
Range assignment (RA) problem, 73, 80, 81
Reactive routing protocols, 44
ReInForM protocol, 54
Relative neighbor graph (RNG), 82
Reliable multi-segment transport (RMST) protocol, 56
Residual energy aware dynamic (READ) topology control algorithm, 100
Right hand rule, 47
Rumor routing, 45

S
Scalable energy-efficient location-aided routing (SElar) protocol, 49
Selective repeat protocol, 32
Self-organizing medium access control for sensor networks (SMACS) protocol, 29
Sensing models, 18
Sensor MAC (S-MAC) protocol, 27
Sensor protocols for information via negotiation (SPIN), 45
Sensor-data management protocol, 184
Simulation event log, 194
Sliding-window protocols, 32
sound reflections, 3
SPAN algorithm, 111
Spanner, 66
State transition matrix, 16
Static global energy-based topology rotation – SGETRot, 135
Static global time-based topology rotation – SGTTRot, 134
Static topology maintenance techniques, 126, 133
STCP protocol, 57
Stop-and-wait protocol, 32
Symmetric range assignment (SRA) problem, 81
Synchronized radio wake up methods, 129

T
Table driven routing protocols, 43
Taxonomy of medium access control protocols, 24
Taxonomy of routing protocols for wireless sensor networks, 43
Time-based triggering criterion, 127
Timeout-MAC (T-MAC) protocol, 27
Topology and energy control algorithm (TECA) for wireless sensor networks, 121
Topology aware routing protocols, 42
Topology construction, 61
Topology control, 61, 67
Topology maintenance, 62, 126
Topology maintenance design issues, 128
Topology maintenance triggering criteria, 127
Topology management by priority ordering (TMPO) protocol, 120
Topology of the network, 66
Traffic-adaptive medium access protocol (TRAMA) protocol, 30
Two radios radio synchronization, 131
Two-ray ground model, 12
Two-state Gilbert error model, 17
Two-state Markov error model, 16

V
Virtual network infrastructures (VNI), 188

W
Weakly symmetric range assignment (WSRA) problem, 81
Wireless sensor MAC (WiseMAC) protocol, 28

X
XTC protocol, 98

Y
Yao graph, 88