Developer Documentation On ONE Simulator

Sahil Gupta
sahilgupta221@gmail.com

Computer Engineering Division
Netaji Subhas Institute of Technology,New Delhi
India

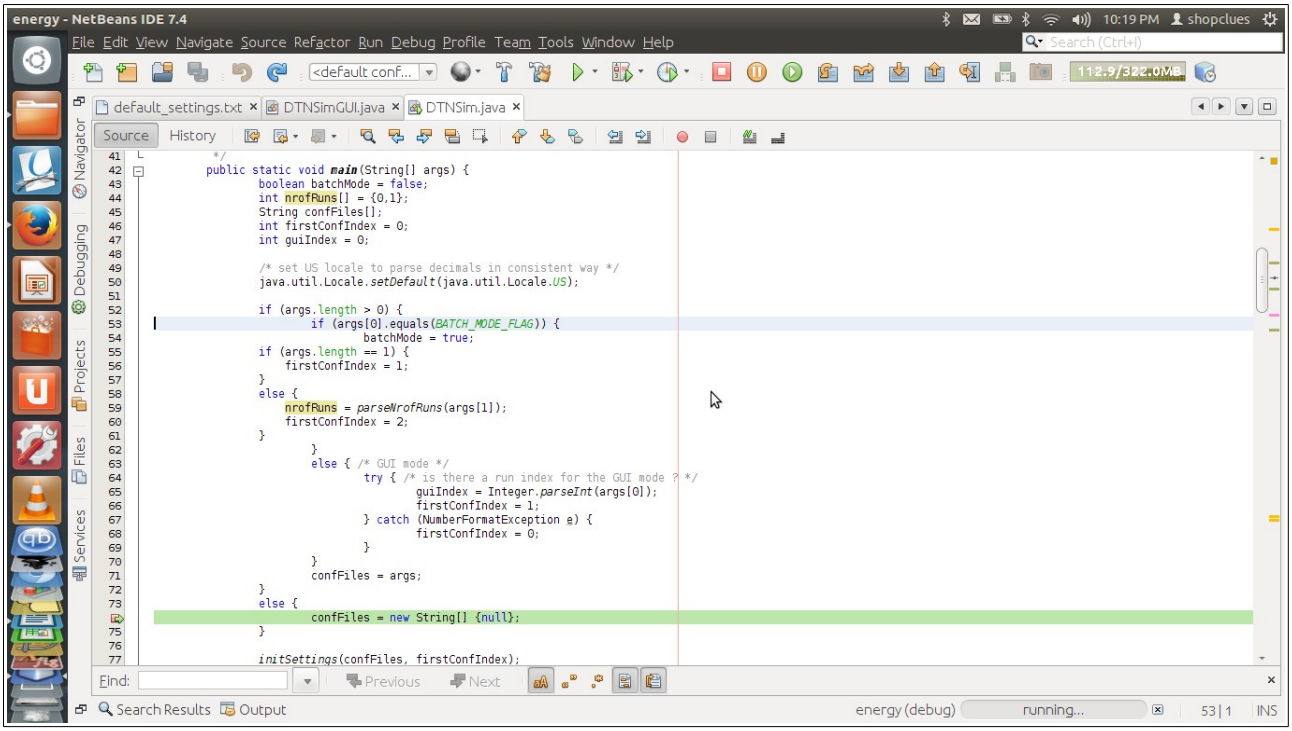
Current Designation:
Software Developer
at
http://www.shopclues.com/

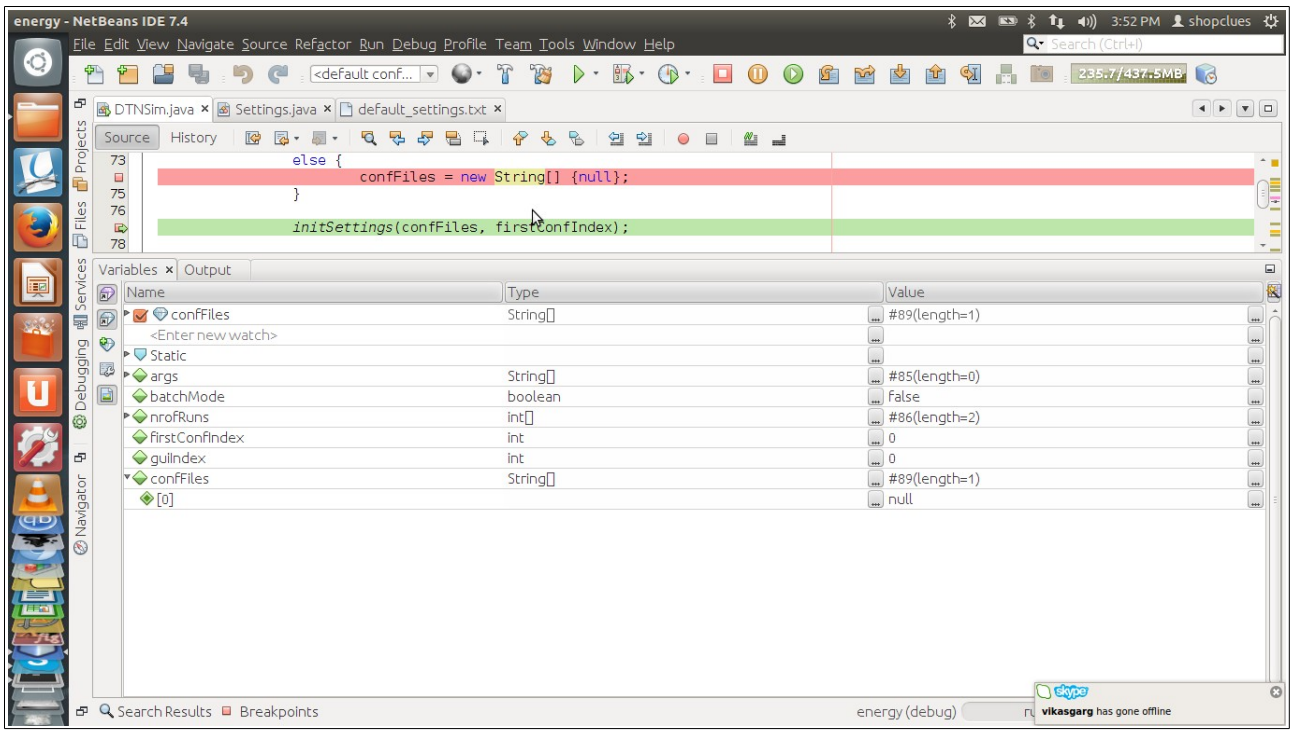
INDEX

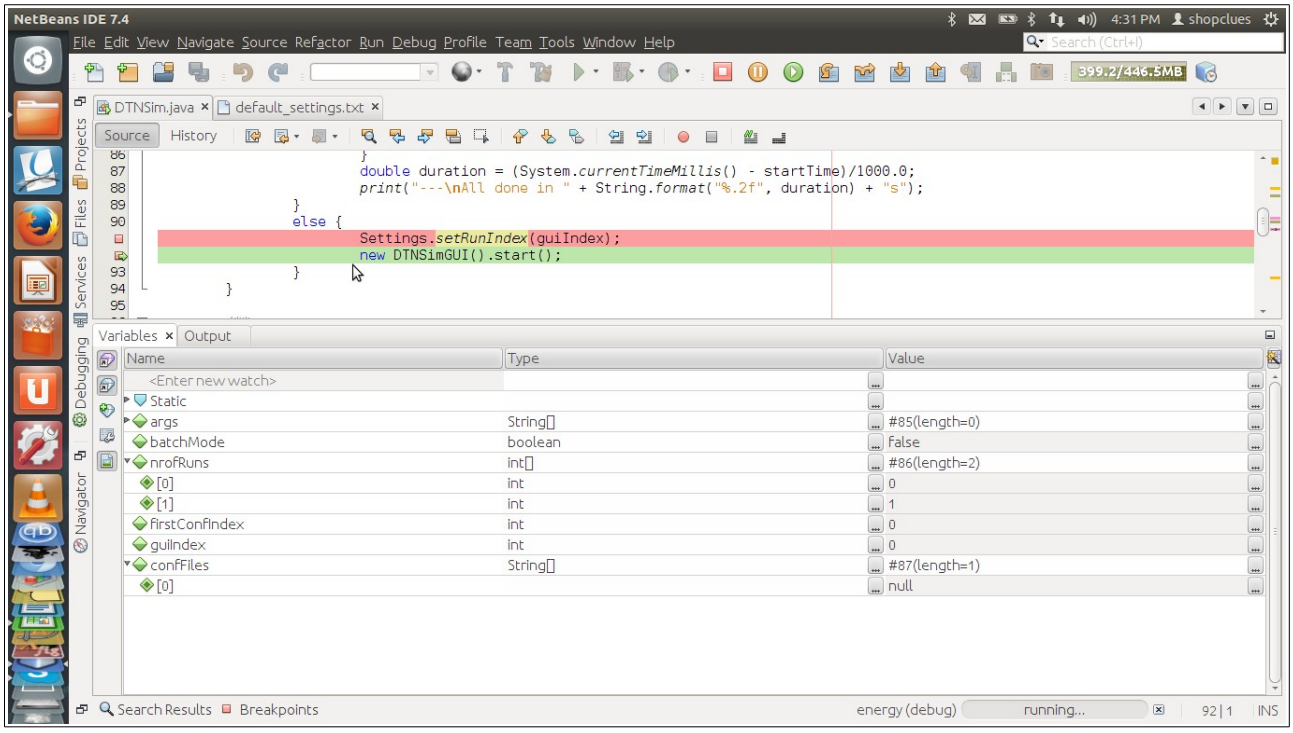
CASE I: Normal Analysis in ONE

Introduction

Normal Analysis can be understood by seeing the Appendix default_file setting section. We have taken Prophet as router and Random Way Point as Mobility Model. This documentation aims to provide the explanation of ONE Architechture with the specific case studies of the default_setting.txt file as shown in the Appendix. Readers are adviced to first read the author's written documentation of ONE from following link:
http://www.netlab.tkk.fi/tutkimus/dtn/theone/pub/the_one.pdf
This documentation also explains various components along with their functionality and responsibility code by code. This Study is divided into two phases
1. Initialisation of Simulation enviroment.
2. Running during simulation till end.



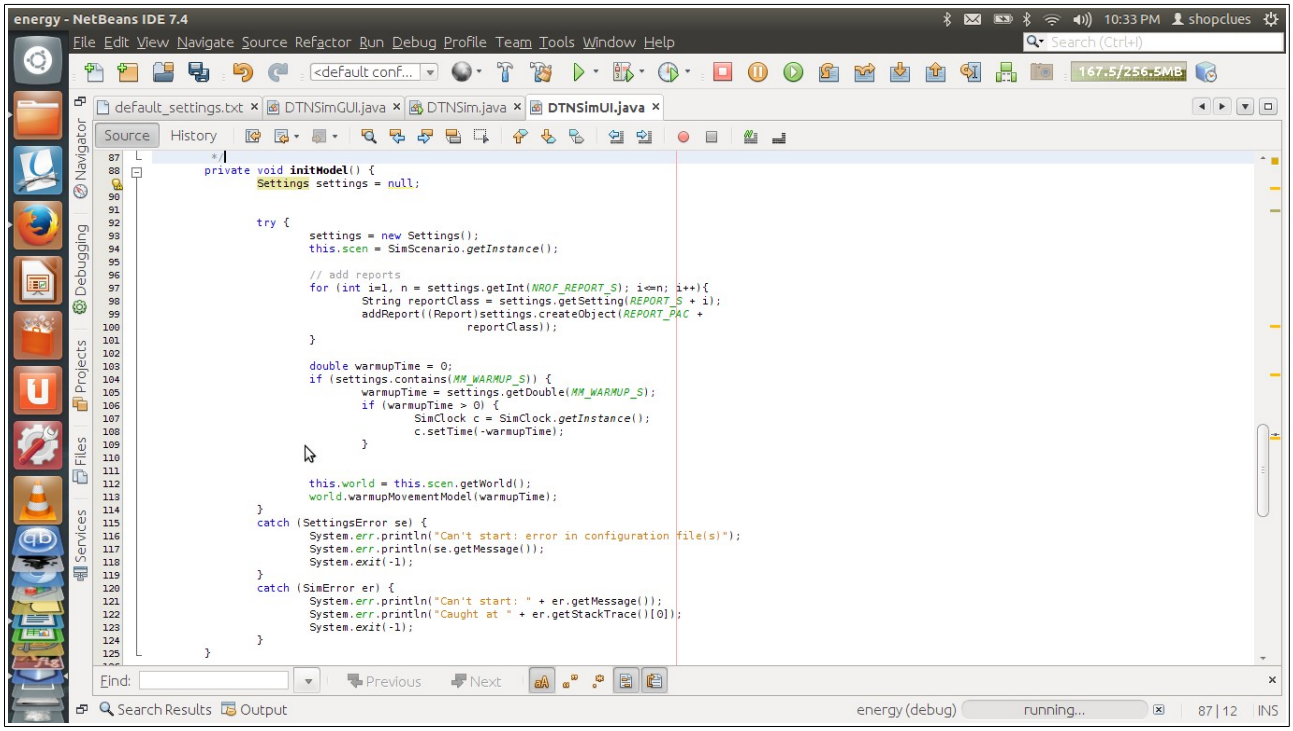




Initialisation Phase working

I am discussing Simulator in GUI mode operation. It starts from DTNSim class. Here you will find main class of simulator. It reaches else part *if(arg.length>0)* condition as seen in green area figure. Here initSettings(confFiles, firstConfIndex) is doing noting. Then it moves to else part of *if (batchMode)* condition. Here *Settings.setRunIndex(guiIndex)* sets the index in setting file. Since there is no addtional config file to read, this function plays no role here. Now *new DTNSimGUI().start()* is the point of interest. It invokes two methods initModel() and runSim(). First one is the topic of disscussion in this section. Running Phase will be covered later on.

```
        /**
         * Starts the simulation.
         */
        public void start() {
                initModel();
                runSim();
        }
```



Code is divided into three parts for discussion of their functionality.
First :
settings = new Settings();
this.scen = SimScenario.getInstance();

Second :
// add reports
for (int i=1, n = settings.getInt(NROF_REPORT_S); i<=n; i++){
String reportClass = settings.getSetting(REPORT_S + i);
addReport((Report)settings.createObject(REPORT_PAC + reportClass));
}

Third:
double warmupTime = 0;
if (settings.contains(MM_WARMUP_S)) {
warmupTime = settings.getDouble(MM_WARMUP_S);
if (warmupTime > 0) {
SimClock c = SimClock.getInstance();

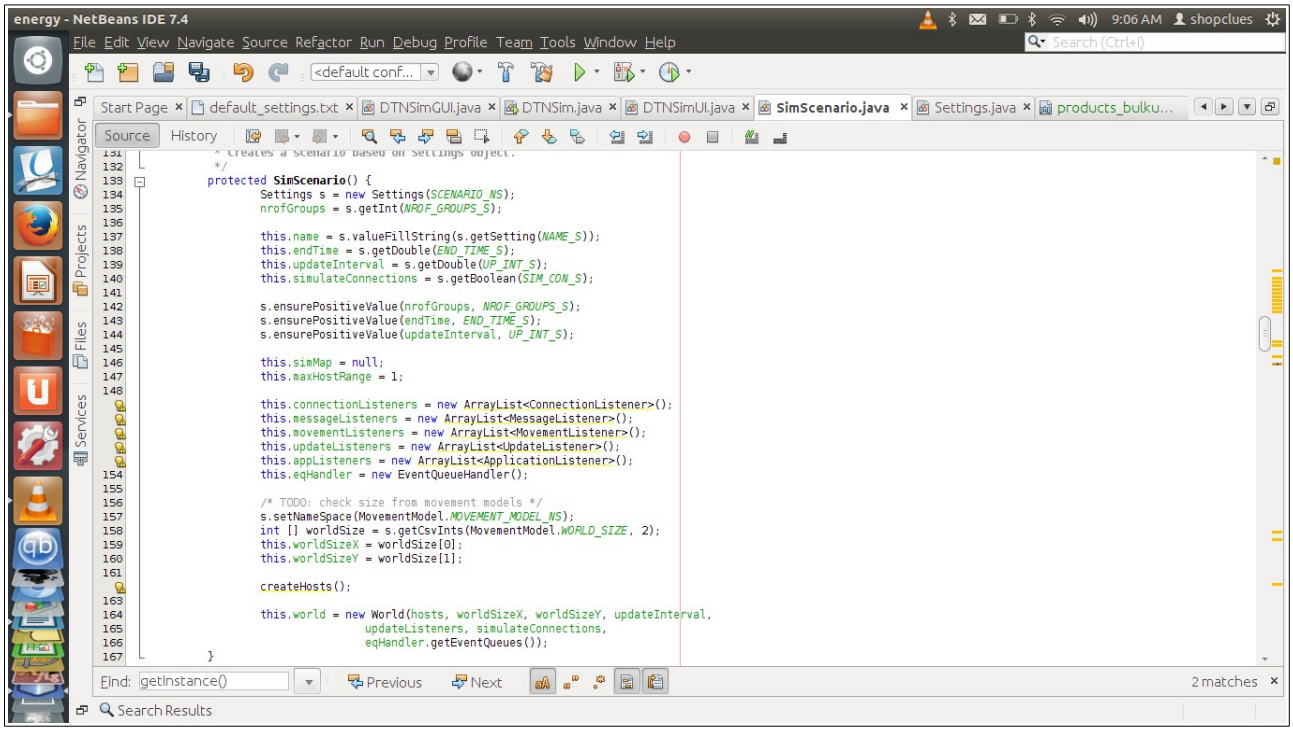
```
        c.setTime(-warmupTime);
        }
}
```

this.world = this.scen.getWorld();
world.warmupMovementModel(warmupTime);

Lets us see the internal working of each task one by one in detail.
settings = new Settings() creates setting object. Setting object is used to read the value of parameters from Config files or generating object of class dynamically. The Setting.java method and atributes can be studied from Docs. This class can be specifically used when you want to obtain value from config file. This can be interger, decimal,string and even can be used to load class object dynamically.
After this, *this.scen = SimScenario.getInstance()* does the major task. Let us see what happens inside this method(see below snapshot). This will call the constructure of Simscenario class.

```
public static SimScenario getInstance() {
        if (myinstance == null) {
                myinstance = new SimScenario();
        }
        return myinstance;
}
```

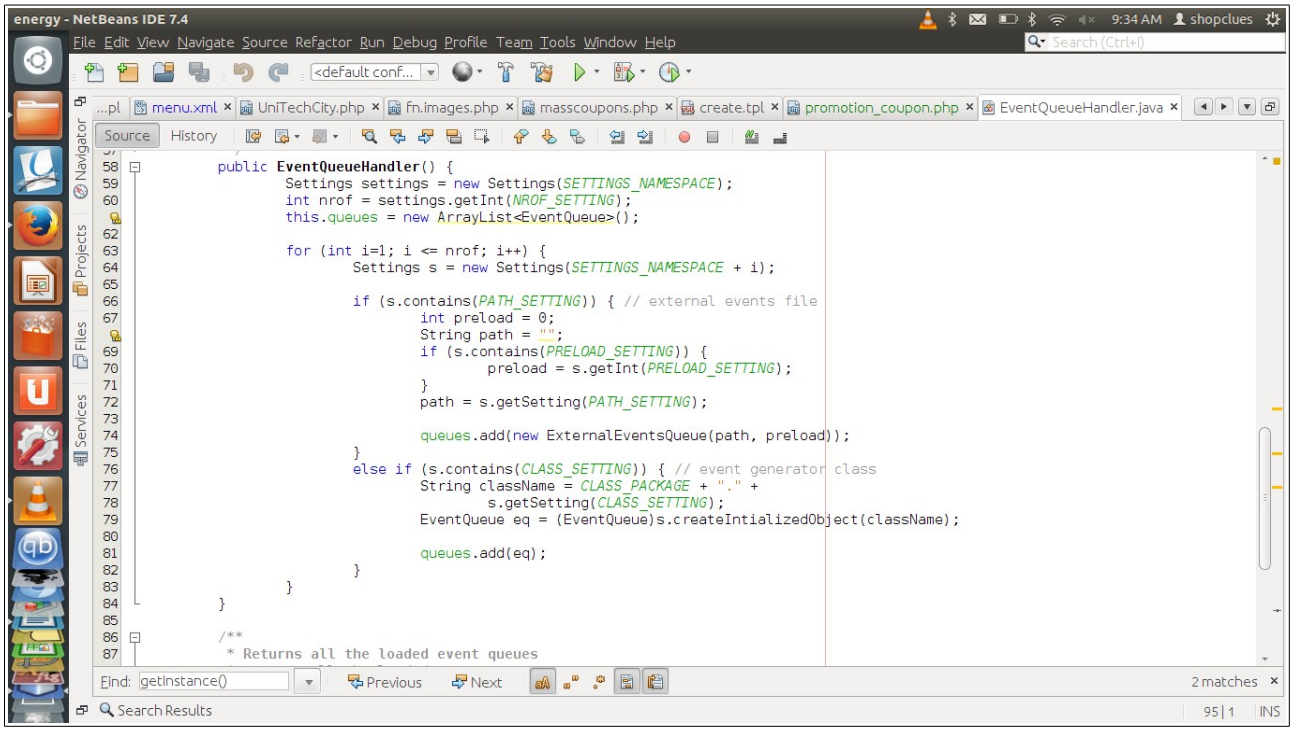
Now flow moves to constructure where scenario elements are created and scenario variables are intialized.



Bunch of stuff is happening here. Let us see one by one.

```
Settings s = new Settings(SCENARIO_NS);
nrofGroups = s.getInt(NROF_GROUPS_S);

this.name = s.valueFillString(s.getSetting(NAME_S));
this.endTime = s.getDouble(END_TIME_S);
this.updateInterval = s.getDouble(UP_INT_S);
this.simulateConnections = s.getBoolean(SIM_CON_S);

s.ensurePositiveValue(nrofGroups, NROF_GROUPS_S);
s.ensurePositiveValue(endTime, END_TIME_S);
s.ensurePositiveValue(updateInterval, UP_INT_S);

this.simMap = null;
this.maxHostRange = 1;

this.connectionListeners = new ArrayList<ConnectionListener>();
this.messageListeners = new ArrayList<MessageListener>();
this.movementListeners = new ArrayList<MovementListener>();
this.updateListeners = new ArrayList<UpdateListener>();
this.appListeners = new ArrayList<ApplicationListener>();
this.eqHandler = new EventQueueHandler();
```

Much of the stuff is self explanatory. Various parameters are getting there values from config file. Various listeners are being intialised. But *this.eqHandler = new EventQueueHandler()* is of special interest to us. Let us see what happens in there.



Clearly it performs two tasks. One External Event Path and Other is event generator intialisation. In

our case there is no external event file and Event generator class is MessageEventGenerator. This will be explained later in running phase the detail function of event generators. Here it is sufficient to know that this class will help in creating message for random source and destination. It also sets message in source node message buffer.

Back to Simscenario constructure.

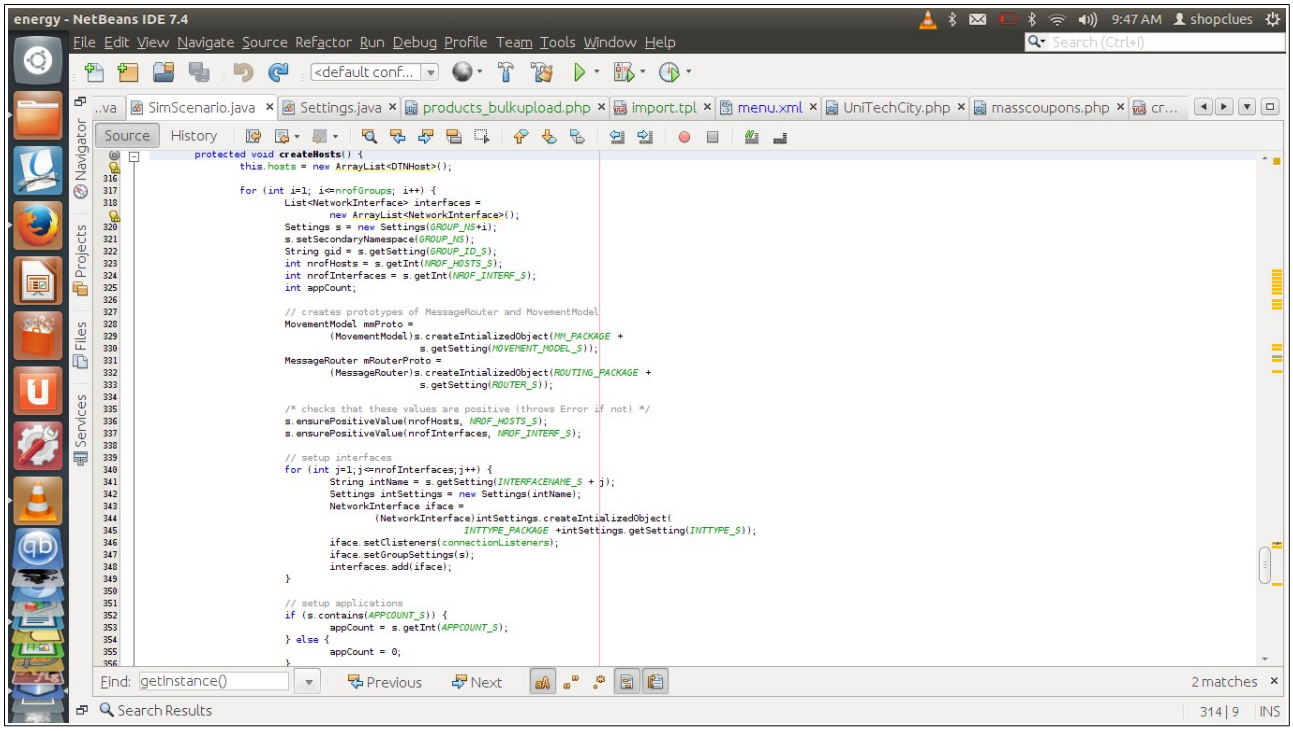
s.setNameSpace(MovementModel.MOVEMENT_MODEL_NS);
int [] worldSize = s.getCsvInts(MovementModel.WORLD_SIZE, 2);
this.worldSizeX = worldSize[0];
this.worldSizeY = worldSize[1];

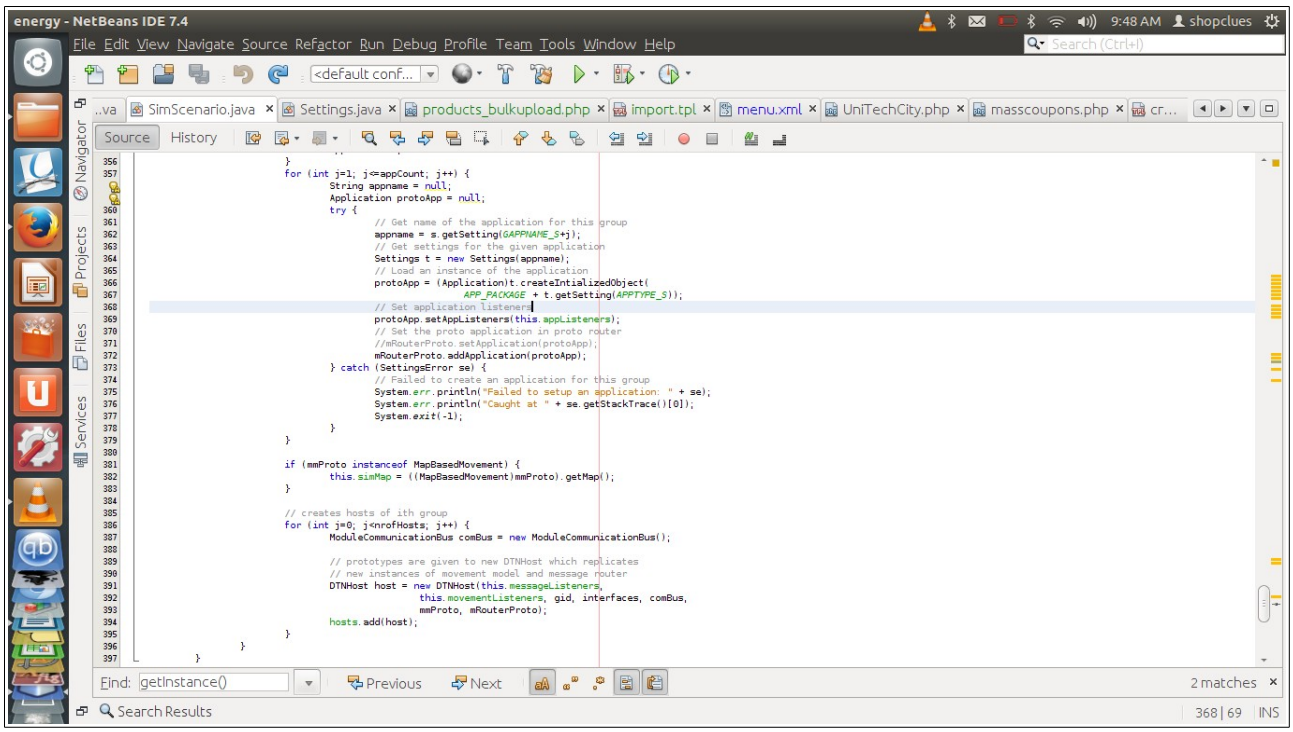
Here World size is getting set.

createHosts();

this.world = new World(hosts, worldSizeX, worldSizeY, updateInterval,
updateListeners, simulateConnections,
eqHandler.getEventQueues());

Both portions are of special interest to us. Let us see first what createHosts() will do first.





Now here something happens that might interest the fellow reasearchers.
Story itself contained in for loop.
for (int i=1; i<=nrofGroups; i++) {
....
...
}

Here nodes are intialised as DTNHost class and all setting associated with it. Here Movement model, Router, Network Interfaces for communication, Application layer(if any) and module communication bus are initialised. The following code sets our nodes of scenario in the enviroment.

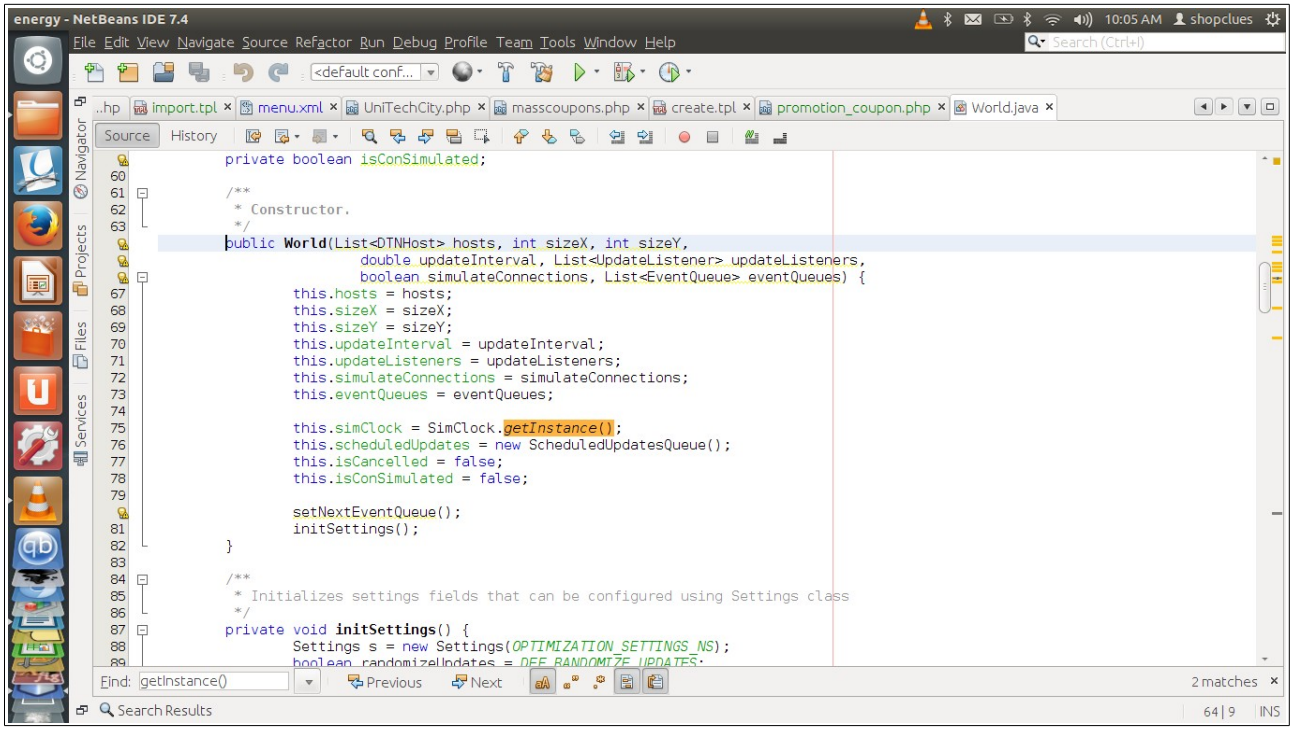
```
DTNHost host = new DTNHost(this.messageListeners,
                this.movementListeners,gid, interfaces, comBus,
                mmProto, mRouterProto);
hosts.add(host);
```

The details of this constructure calling and network interface position updatation working in the form of cells will be disscussed in running phase at its appropriate position.

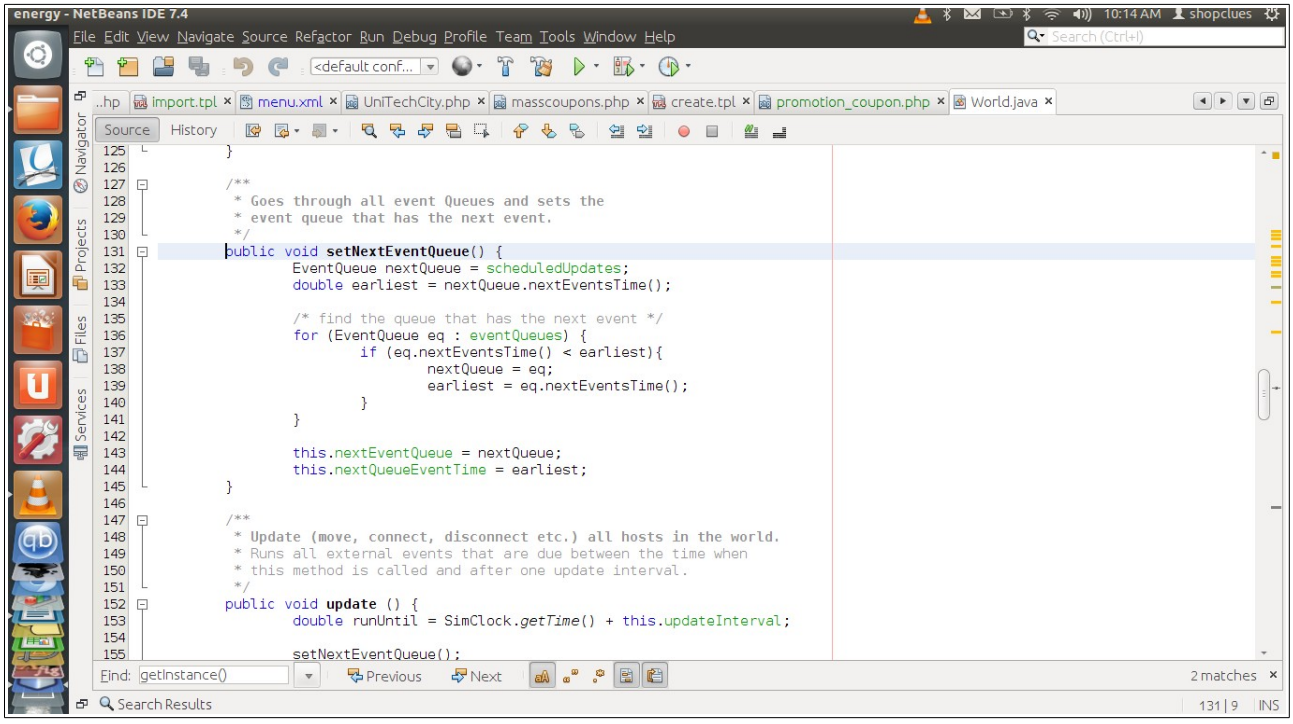
Coming back to

```
createHosts();

this.world = new World(hosts, worldSizeX, worldSizeY, updateInterval,
                updateListeners, simulateConnections,
                eqHandler.getEventQueues());
```

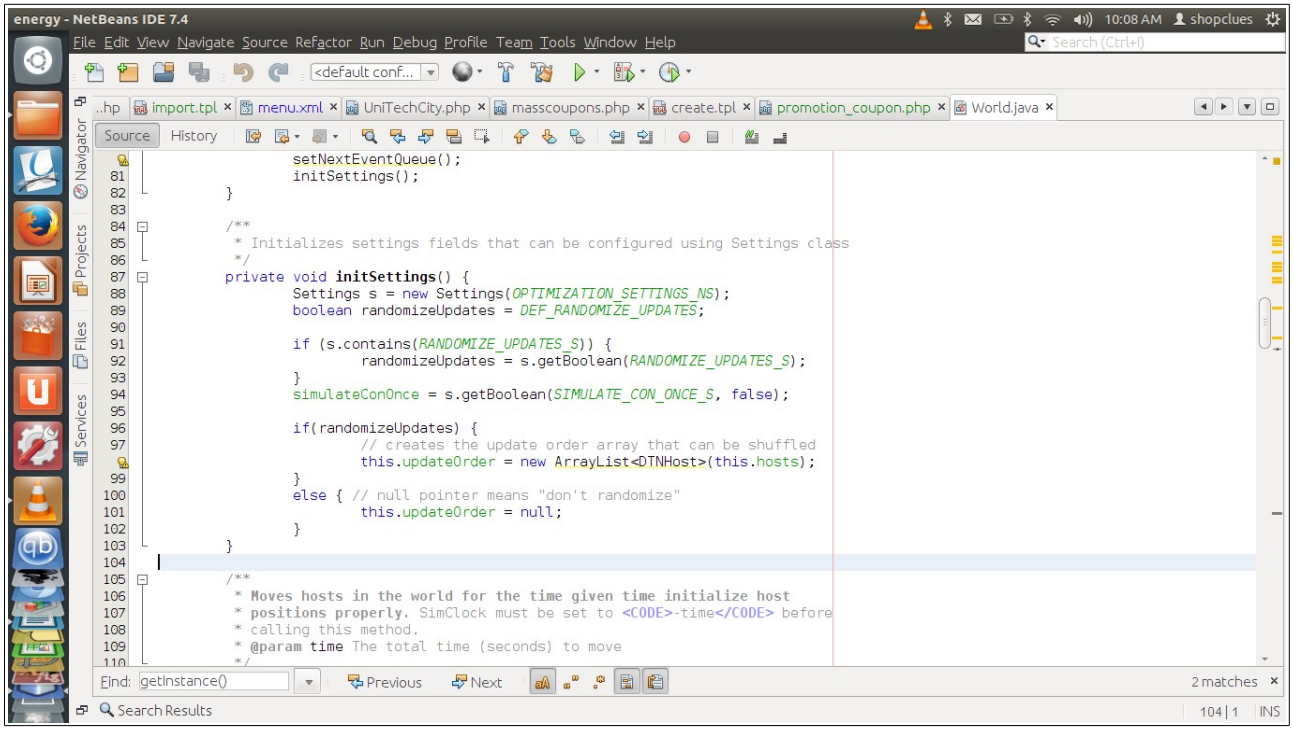
Now the world is intialised which is used in running phase to update nodes and carrying out the simulation. Lets have a look inside its constructure.



Everything is self explanatory. The *setNextEventQueue()* sets the eventQueues which is MessageEventGenerator class in our case. Let us see how it looks.



The last line *initSettings()* sets the order in which nodes will be updated. It can be Random or in accordance to network address of nodes according to the config file setting. Let us see how this looks like.

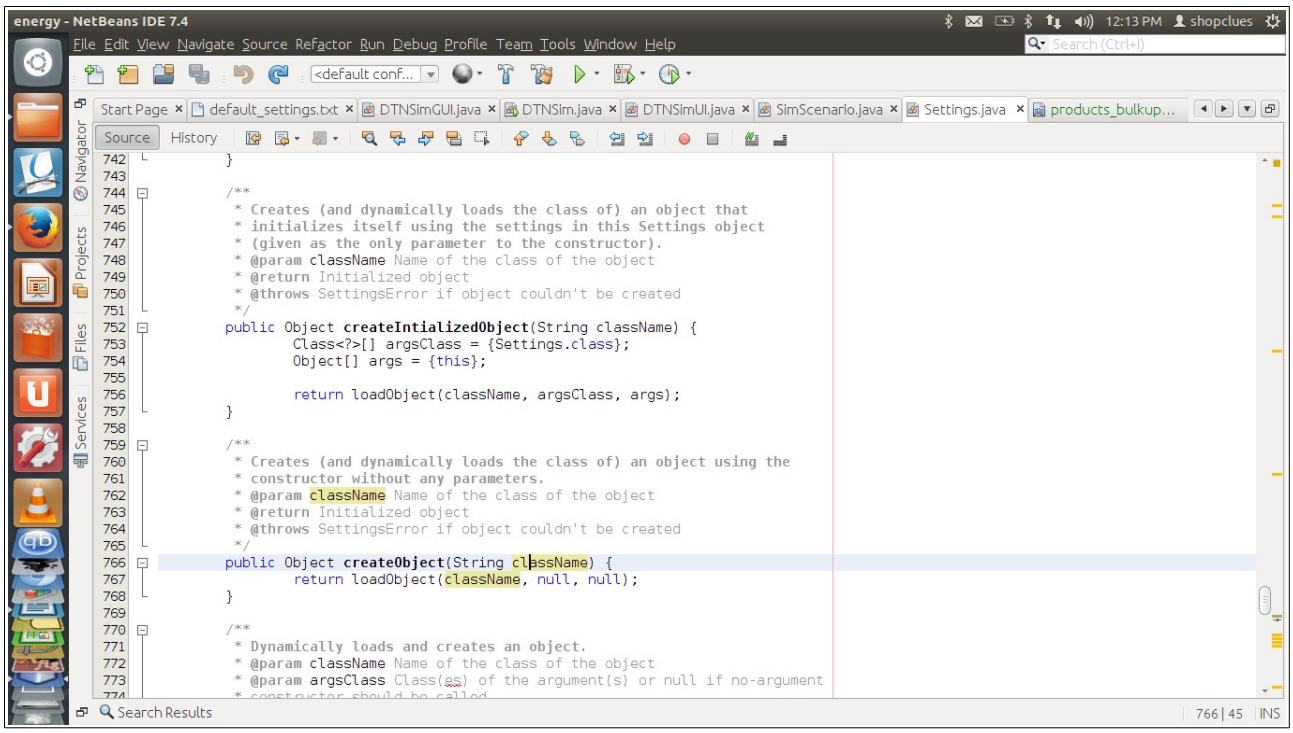


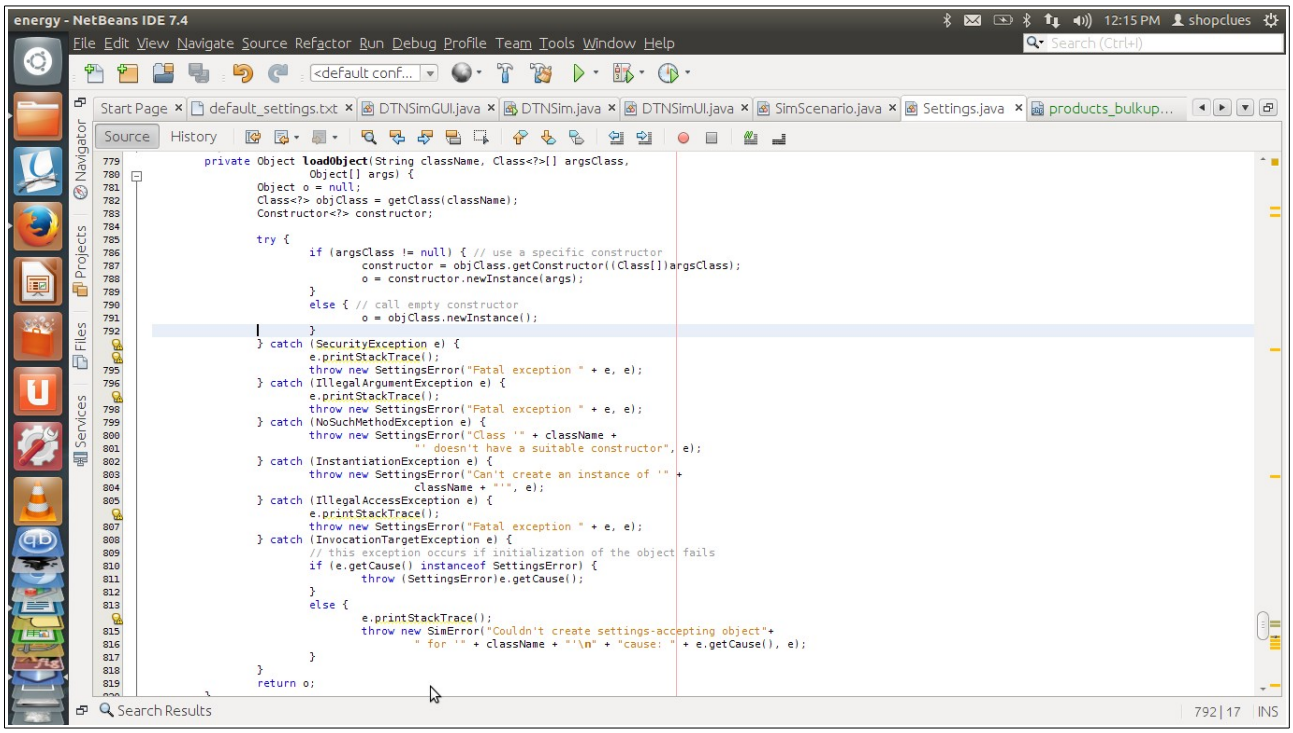
Comming back to DTNSimUI *initModel()* method.

Lets us see what task second will do.

// add reports
for (int i=1, n = settings.getInt(NROF_REPORT_S); i<=n; i++){
String reportClass = settings.getSetting(REPORT_S + i);
addReport((Report)settings.createObject(REPORT_PAC + reportClass));
}

Here three things will happen. First dynamic object loading through createObject method. Then typecasting into Report Class and then passing it to addReport method. Let us see them one by one.





Here createIntializedObject and createObject methods are used to load the object dynamically. This further use loadObject Method. Class<?> is the key here for loading object dynamically. Readers are adviced to see Oracle Java Docs for further information on this topic.

Now see addReport Method in DTNSimUI class.

```
/**
```

```
	 * Adds a new report for simulator
	 * @param r Report to add
	 */
	protected void addReport(Report r) {
		if (r instanceof MessageListener) {
			scen.addMessageListener((MessageListener)r);
		}
		if (r instanceof ConnectionListener) {
			scen.addConnectionListener((ConnectionListener)r);
		}
		if (r instanceof MovementListener) {
			scen.addMovementListener((MovementListener)r);
		}
		if (r instanceof UpdateListener) {
			scen.addUpdateListener((UpdateListener)r);
		}
		if (r instanceof ApplicationListener) {
			scen.addApplicationListener((ApplicationListener)r);
		}
		this.reports.add(r);
	}
}
```

Here you get your first clue how various reports in report Pakage generate report. Funda is pretty simple. Reports implement various kinds of interfaces depending upon which method they want to invoke or define. Later in running phase we will see that various classes get iterated in their interface loop like MessageListeners UpdateListener,MovementListener etc. This is again basic java stuff and to get indepth knowledge of such schemes, One can read Design Patterns in Java Head First Edition or any other book on Design Patterns can be reffered.

Moving on to the third task.

```
double warmupTime = 0;
if (settings.contains(MM_WARMUP_S)) {
	warmupTime = settings.getDouble(MM_WARMUP_S);
	if (warmupTime > 0) {
		SimClock c = SimClock.getInstance();
		c.setTime(-warmupTime);
	}
}

this.world = this.scen.getWorld();
world.warmupMovementModel(warmupTime);
```

Here, First Warmup time is obtained. Then set in the simulator clock. Then world warmupMovementModel Method is called.

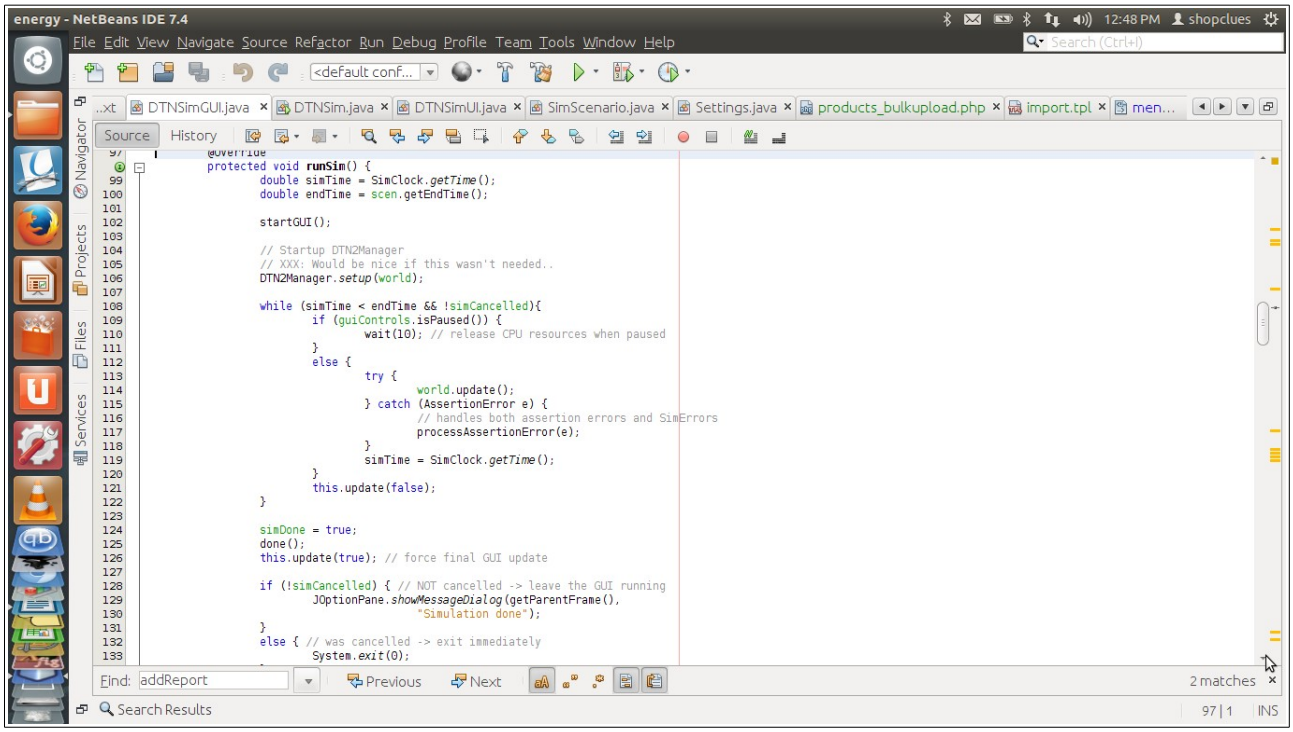
```
/**
 * Moves hosts in the world for the time given time initialize host
 * positions properly. SimClock must be set to <CODE>-time</CODE> before
 * calling this method.
 * @param time The total time (seconds) to move
 */
public void warmupMovementModel(double time) {
        if (time <= 0) {
                return;
        }

        while(SimClock.getTime() < -updateInterval) {
                moveHosts(updateInterval);
                simClock.advance(updateInterval);
        }

        double finalStep = -SimClock.getTime();

        moveHosts(finalStep);
        simClock.setTime(0);
}
```

moveHost method will be discussed in the runnning phase section.

Now again back to start() Method of DTNSimUI class. Till now all required objects are loaded and are interlinked as seen above. Now as the time passes simulator call methods on UpdateInterval value(as defined in config file). Let us see Running Phase.

```
/**
 * Starts the simulation.
 */
public void start() {
        initModel();
        runSim();
}
```

Running Phase working



The run sim code has been divided into following section for discussion:

First:

```
double simTime = SimClock.getTime();
double endTime = scen.getEndTime();

startGUI();

// Startup DTN2Manager
// XXX: Would be nice if this wasn't needed..
DTN2Manager.setup(world);
```

Second:

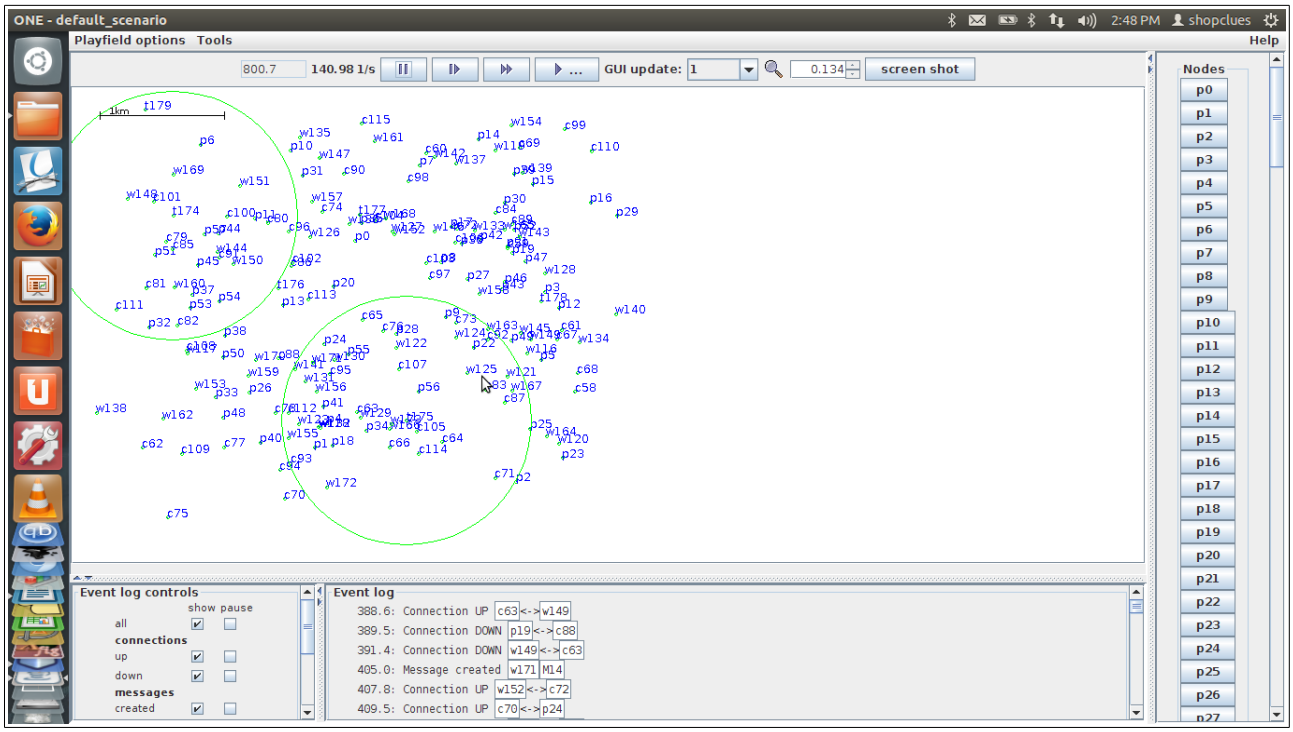
```
while (simTime < endTime && !simCancelled){
        if (guiControls.isPaused()) {
                wait(10); // release CPU resources when paused
        }
        else {
                try {
                        world.update();
                } catch (AssertionError e) {
                        // handles both assertion errors and SimErrors
                        processAssertionError(e);
                }
```

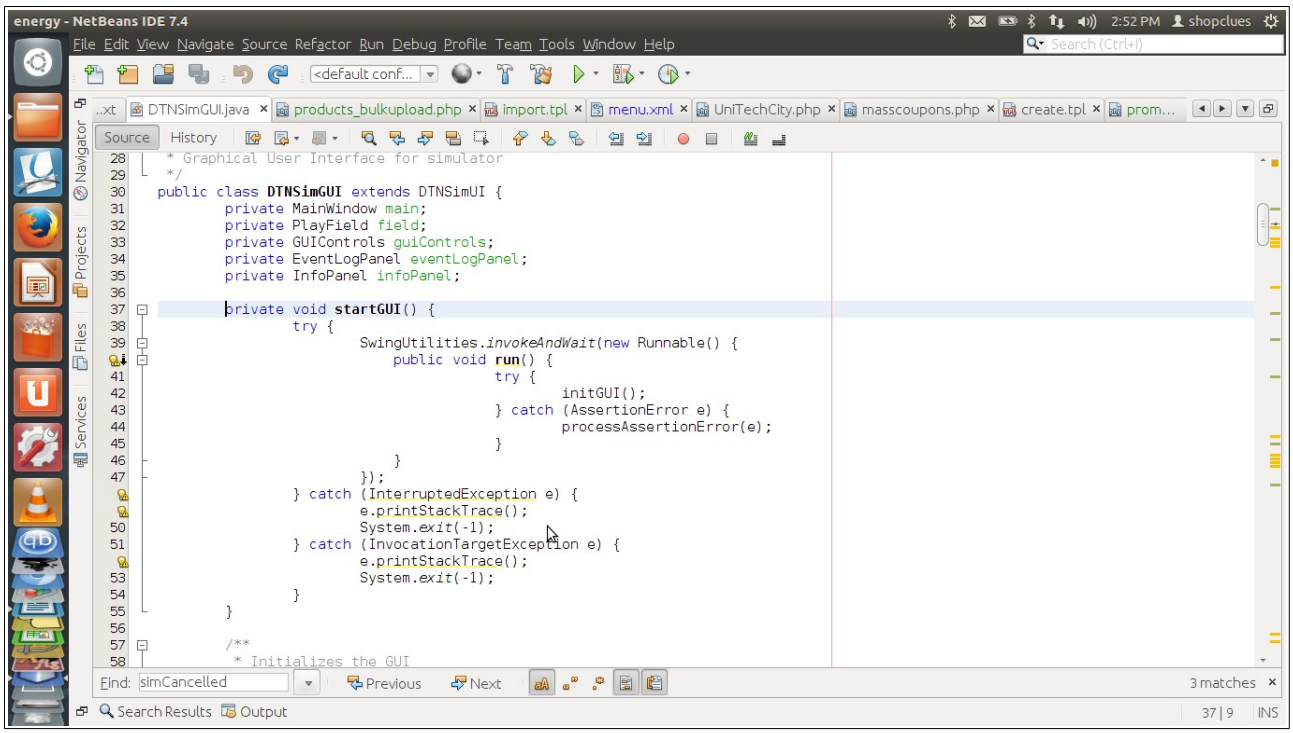
```
                simTime = SimClock.getTime();
            }
            this.update(false);
        }
```

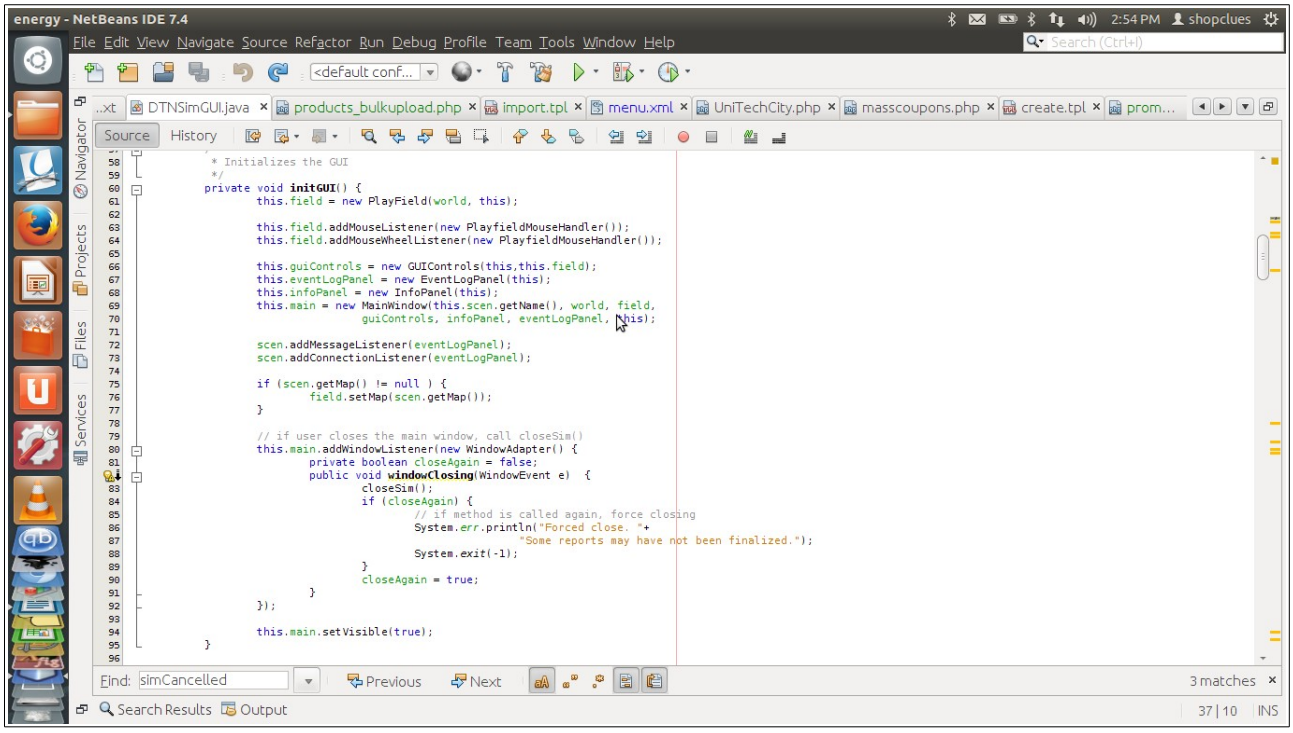
Third:

```
        simDone = true;
        done();
        this.update(true); // force final GUI update

        if (!simCancelled) { // NOT cancelled -> leave the GUI running
            JOptionPane.showMessageDialog(getParentFrame(),
                    "Simulation done");
        }
        else { // was cancelled -> exit immediately
            System.exit(0);
        }
```

In first task, simulator time and end time is obtained. *StartGUI()* method will initialise the GUI interface. See the figure. Various Pannels in the main frame can be seen. Further snapshot shows flow of code. Basics of Java frames and Jpannel etc is sufficient to explore futher GUI component.





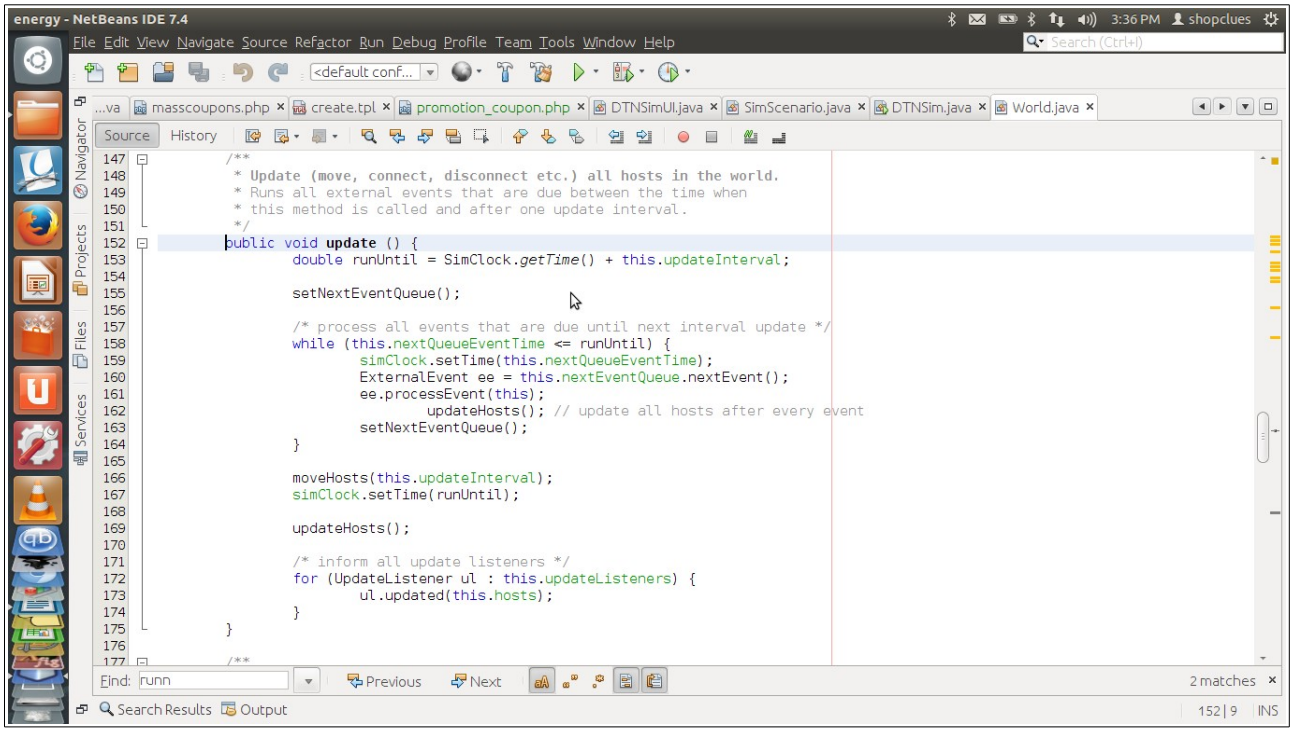


In our case *DTN2Manager.setup(world)* is not required. Now see second task.

In second task GUI components are having the same meaning as their names. While loop will run till endtime time is reached and update functions will call every updateInterval value from config file. Now if pause button is not pressed it enters the else part of *if (guiControls.isPaused())* then *world.update()* method is called. This world.update method is of pretty much interest to Our DTN research Community.
Real Game begins from now. In this method Events are called, Hosts are updated, Routers are

updated and Hosts are moved to new incremental location as per mobility model choosen from config. Here connections are also made and broke. Get a cup of coffee now. You need to start this section from fresh look. Here the implementations of most of the research problems lie like one way connections or connection manipulation, Queing policy, routing design, mobility design etc. Most of the people are interested in such kind of implementations as per the experience. Here we go.

Here we start with the update method of world.



Here we again divide the code in three as follows:

First:

```
double runUntil = SimClock.getTime() + this.updateInterval;
setNextEventQueue();
```

Second:

```
while (this.nextQueueEventTime <= runUntil) {
    simClock.setTime(this.nextQueueEventTime);
    ExternalEvent ee = this.nextEventQueue.nextEvent();
    ee.processEvent(this);
  updateHosts(); // update all hosts after every event
    setNextEventQueue();
}
```

Third:

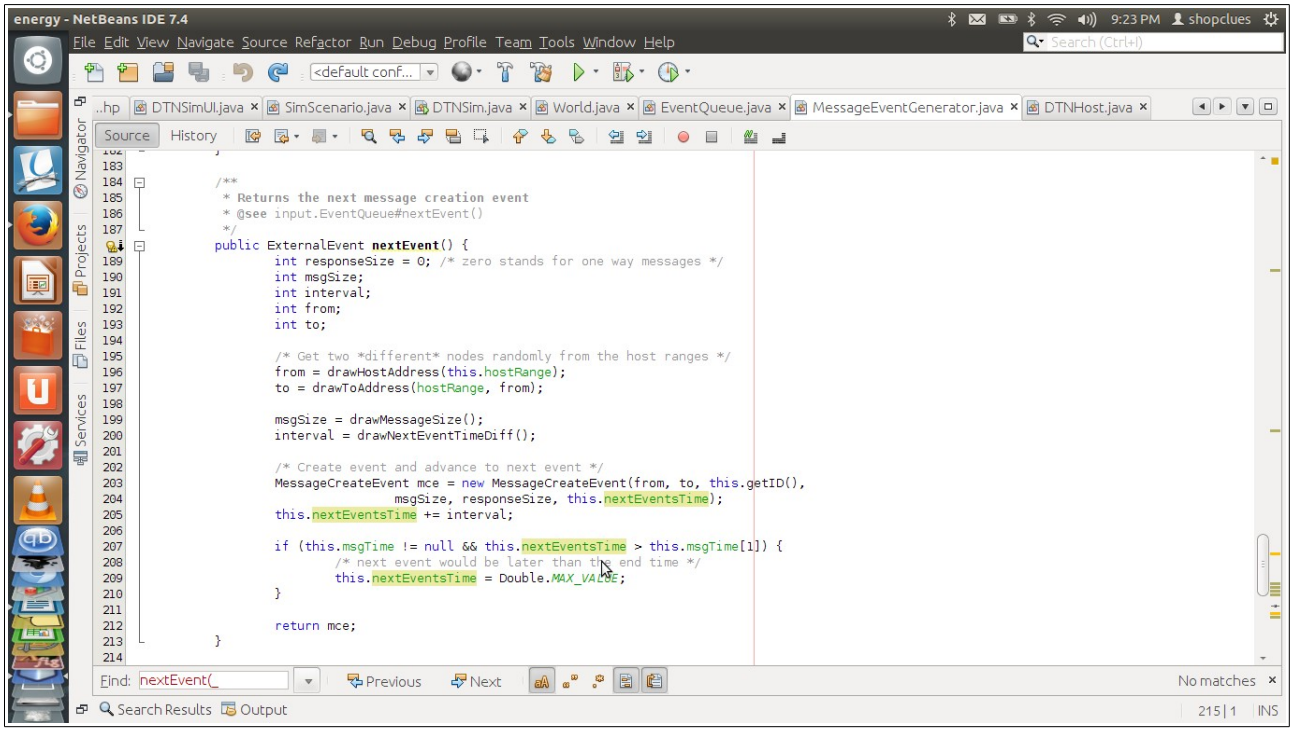
```
moveHosts(this.updateInterval);
simClock.setTime(runUntil);
updateHosts();
/* inform all update listeners */
for (UpdateListener ul : this.updateListeners) {
    ul.updated(this.hosts);
```

}

Lets us discuss it one by one.

```
        /**
         * Goes through all event Queues and sets the
         * event queue that has the next event.
         */
        public void setNextEventQueue() {
                EventQueue nextQueue = scheduledUpdates;
                double earliest = nextQueue.nextEventsTime();

                /* find the queue that has the next event */
                for (EventQueue eq : eventQueues) {
                        if (eq.nextEventsTime() < earliest){
                                nextQueue = eq;
                                earliest = eq.nextEventsTime();
                        }
                }

                this.nextEventQueue = nextQueue;
                this.nextQueueEventTime = earliest;
        }
```

Here it set next queue that has next event to be executed. In our case it will be only MessageEventGenerator class. Here nextEventsTime method will return random time between Events1.interval = 25,35 value in config file.



In second task, Simulator clock sets time upto *nextQueueEventTime* value.
Here, *this.nextEventQueue* is MessageEventGenerator. The nextEvent Method of

MessageEventGenerator is shown above.

Disscussion on nextEvent Method of MessageEventGenerator:
Parameters like r*esponseSize*, *msgSize*, *interval*, *from* and *to* obtained their value. Now this line
MessageCreateEvent mce = new MessageCreateEvent(from, to, this.getID(),
msgSize, responseSize, this.nextEventsTime);

creates the MessageCreateEvent object which is required in world.update method for process events.

```
        public MessageCreateEvent(int from, int to, String id, int size,
                        int responseSize, double time) {
                super(from,to, id, time);
                this.size = size;
                this.responseSize = responseSize;
        }
```

Now back to update method. Line *ee.processEvent(this)* calls the processEvent Method of MessageCreateEvent which creates Message and put it in the source message buffer.

```
        /**
         * Creates the message this event represents.
         */
        @Override
        public void processEvent(World world) {
                DTNHost to = world.getNodeByAddress(this.toAddr);
                DTNHost from = world.getNodeByAddress(this.fromAddr);

                Message m = new Message(from, to, this.id, this.size);
                m.setResponseSize(this.responseSize);
                from.createNewMessage(m);
        }
```

Back to world.update Method. It is time to look at updateHosts Method of world class.

```
177         /**
178          * Updates all hosts (calls update for every one of them). If update
179          * order randomizing is on (updateOrder array is defined), the calls
180          * are made in random order.
181          */
182         private void updateHosts() {
183                 if (this.updateOrder == null) { // randomizing is off
184                         for (int i=0, n = hosts.size();i < n; i++) {
185                                 if (this.isCancelled) {
186                                         break;
187                                 }
188                                 hosts.get(i).update(simulateConnections);
189                         }
190                 }
191                 else { // update order randomizing is on
192                         assert this.updateOrder.size() == this.hosts.size() :
193                                 "Nrof hosts has changed unexpectedly";
194                         Random rng = new Random(SimClock.getIntTime());
195                         Collections.shuffle(this.updateOrder, rng);
196                         for (int i=0, n = hosts.size();i < n; i++) {
197                                 if (this.isCancelled) {
198                                         break;
199                                 }
                                  this.updateOrder.get(i).update(simulateConnections);
                          }
202                 }
203
204                 if (simulateConOnce && simulateConnections) {
205                         simulateConnections = false;
206                 }
207         }
```

In our case control reaches to red line. So its obivious it is randomise update in nature. There update method of each node(DTNHost) in scenario is called. Let us see now update method of DTNHost.

```
328
329         /**
330          * Updates node's network layer and router.
331          * @param simulateConnections Should network layer be updated too
332          */
333         public void update(boolean simulateConnections) {
334                 if (!isRadioActive()) {
335                         // Make sure inactive nodes don't have connections
336                         tearDownAllConnections();
337                         return;
338                 }
339
340                 if (simulateConnections) {
341                         for (NetworkInterface i : net) {
342                                 i.update();
343                         }
344                 }
345                 this.router.update();
346         }
```

Meanwhile in DTNHost class.
In its constructure at initialisation phase values of MessageListeners are MessageStatsReport and EventLogPannel. Router is prophet and Network Interface is SimpleBroadcastInterface.

comBus	ModuleCommunicationBus	#2116
destination		null
energy_thrushold_counter	int	0
location	Coord	#2110
movListeners	ArrayList	"size = 0"
movement	RandomWaypoint	#2112
msgListeners	ArrayList	"size = 2"
[0]	MessageStatsReport	#2139
[1]	EventLogPanel	#2140
name	String	"p55"
net	ArrayList	"size = 1"
[0]	SimpleBroadcastInterface	#2137
nextTimeToMove	double	38.72030292546405
path		null
router	ProphetRouter	#2111
speed	double	0.0
total_energy_charged	double	0.0

Back to DTNHost.update Method. *IsRadioActive* method first checks interface at first position of interface list. This returns true or false depending upon whether node is active or not. It is also decide based on reason like config value, Current Energy value and transmission range is 0 or not. See Pics for clarity.

```
120         /**
121          * Returns true if this node's radio is active (false if not)
122          * @return true if this node's radio is active (false if not)
123          */
124         public boolean isRadioActive() {
125                 /* TODO: make this work for multiple interfaces */
126                 return this.getInterface(1).isActive();
127         }
```

```
224         /**
225          * Returns true if the interface is on at the moment (false if not)
226          * @return true if the interface is on at the moment (false if not)
227          */
228         public boolean isActive() {
229                 boolean active = ah.isActive(this.activenessJitterValue);
230
231                 if (active && host.getComBus().getDouble(EnergyModel.ENERGY_VALUE_ID,
232                                 1) <= 0) {
233                         /* TODO: better way to check battery level */
234                         /* no battery -> inactive */
235                         active = false;
236                 }
237
238                 if (active == false && this.transmitRange > 0) {
239                         /* not active -> make range 0 */
240                         this.oldTransmitRange = this.transmitRange;
241                         host.getComBus().updateProperty(RANGE_ID, 0.0);
242                 } else if (active == true && this.transmitRange == 0.0) {
243                         /* active, but range == 0 -> restore range  */
244                         host.getComBus().updateProperty(RANGE_ID,
245                                         this.oldTransmitRange);
246                 }
247                     return active;
248         }
```

In our case it is true.So it will not enter here. The case in which it enters in this section of code *tearDownAllConnections* method will remove all connection assosiated with it. See snapshot of it.

```
        /**
         * Tears down all connections for this host.
         */
        private void tearDownAllConnections() {
                for (NetworkInterface i : net) {
                        // Get all connections for the interface
                        List<Connection> conns = i.getConnections();
                        if (conns.size() == 0) continue;

                        // Destroy all connections
                        List<NetworkInterface> removeList =
                                new ArrayList<NetworkInterface>(conns.size());
                        for (Connection con : conns) {
                                removeList.add(con.getOtherInterface(i));
                        }
                        for (NetworkInterface inf : removeList) {
                                i.destroyConnection(inf);
                        }
                }
        }
```

According to our case study. We reach at the for loop of *NetworkInterface* iteration. Here its update method is called. This will call the update method of SimpleBroadcastInterface.

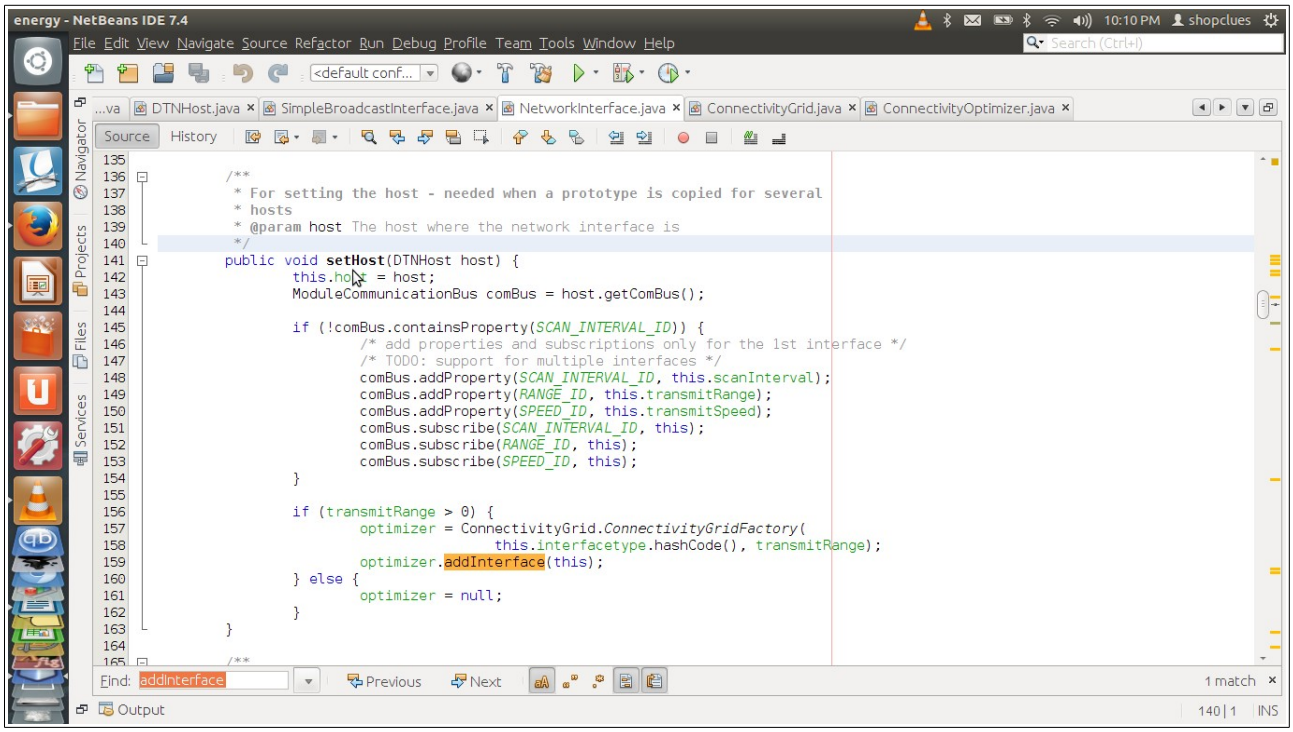
Before proceeding further it is the right time to discuss the working of DTNHost construture from createHost method of Simscenario that we left at initialisation phase and some files associated in its working.

```
public DTNHost(List<MessageListener> msgLs,
               List<MovementListener> movLs,
               String groupId, List<NetworkInterface> interf,
               ModuleCommunicationBus comBus,
               MovementModel mmProto, MessageRouter mRouterProto) {
        this.comBus = comBus;
        this.location = new Coord(0,0);
        this.address = getNextAddress();
        this.name = groupId+address;
        this.net = new ArrayList<NetworkInterface>();

        for (NetworkInterface i : interf) {
                NetworkInterface ni = i.replicate();
                ni.setHost(this);
                net.add(ni);
        }

        // TODO - think about the names of the interfaces and the nodes

        this.msgListeners = msgLs;
        this.movListeners = movLs;

        // create instances by replicating the prototypes
        this.movement = mmProto.replicate();
        this.movement.setComBus(comBus);
        this.movement.setHost(this);
        setRouter(mRouterProto.replicate());

        this.location = movement.getInitialLocation();

        this.nextTimeToMove = movement.nextPathAvailable();
        this.path = null;

        if (movLs != null) { // inform movement listeners about the location
                for (MovementListener l : movLs) {
                        l.initialLocation(this, this.location);
                }
        }
}
```

Every part of code is self explanatory but there is something more interesting going on in the highlighted parts.

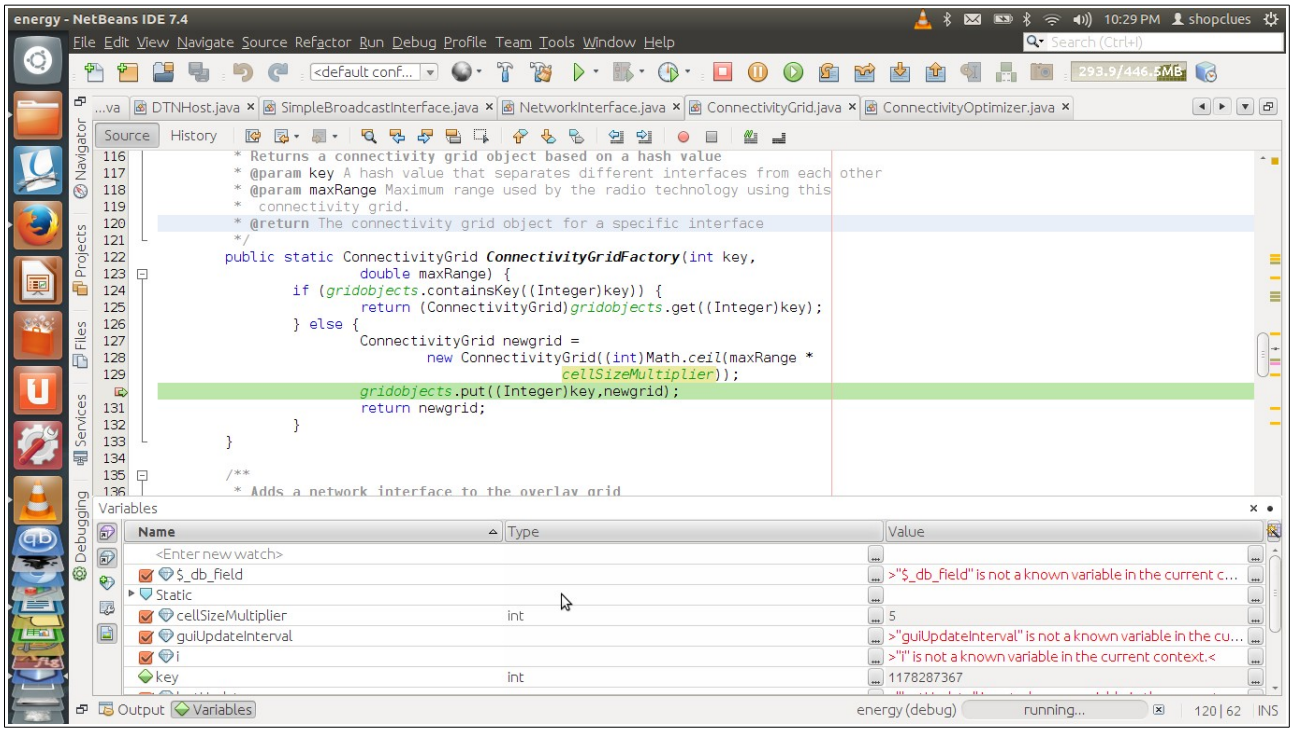
```
for (NetworkInterface i : interf) {
        NetworkInterface ni = i.replicate();
        ni.setHost(this);
        net.add(ni);
}
```

Here ni.setHost method assosiates interface of DTNHost with cell of grid which is based on it transmission range. So if it has n interfaces, then it has ten different topologies of grids in which it is associated. Now let us step by step see what is going on.

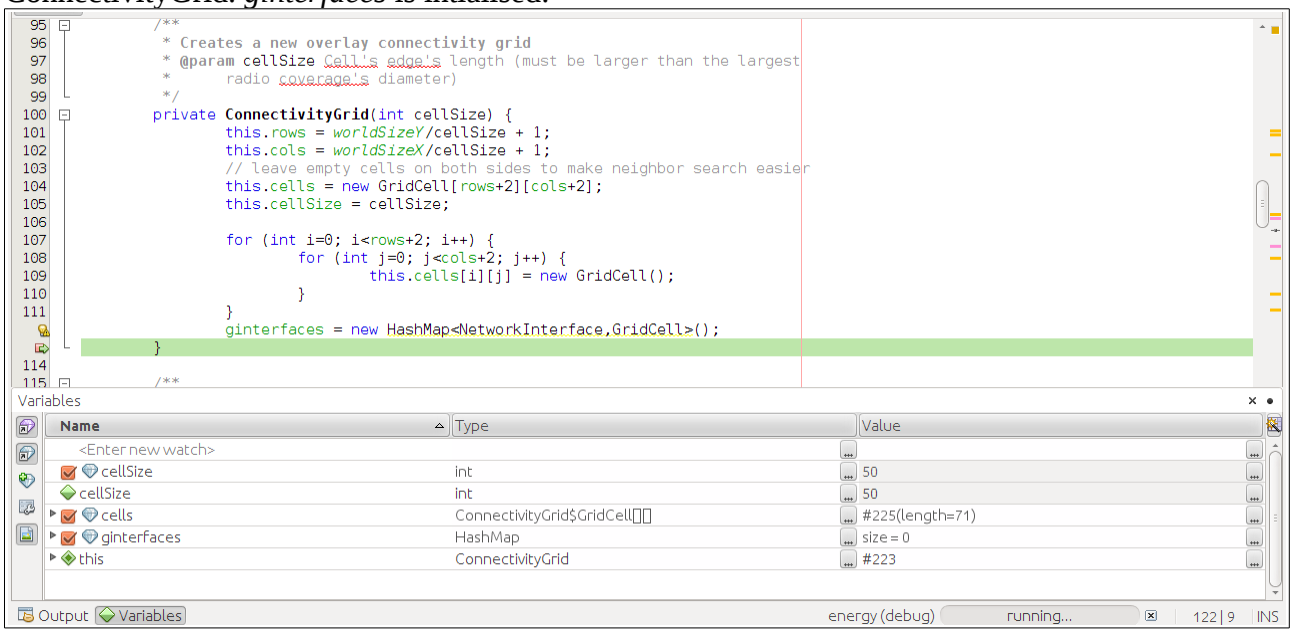


Here First it registers basic properties to communication bus of its DTNHost class. Then optimizer

makes grids based on transmission range and cellSizeMultiplier value.



Lets see what is really happening behind the scence.
Here it makes ConnectivityGrid new object which futher makes rows and column and so the cells array is filled with the new GridCell inner class object. GridCell is the inner class to ConnectivityGrid. *ginterfaces* is intialised.



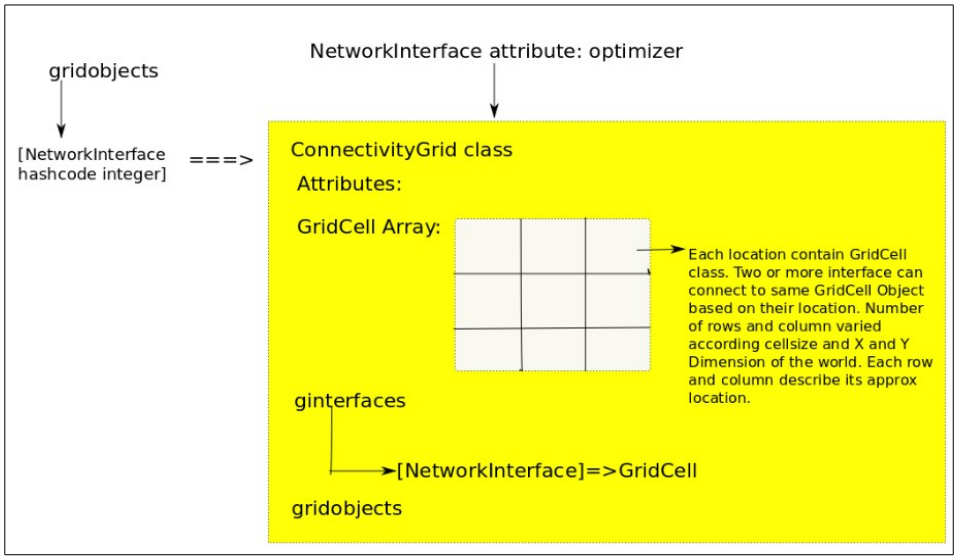
gridobjects.put((Integer)key,newgrid) registers this grid with above mention topology with hashcode of interface in *gridobjects.* This attribute is static and so the method in which all things are happening. So it is a global variable by nature and available to all.

Comming back to NetworkInterface.setHost method next line:
optimizer.addInterface(this);

addinterface associates cell with interface and interface with cell.

```
134
135         /**
136          * Adds a network interface to the overlay grid
137          * @param ni The new network interface
138          */
139         public void addInterface(NetworkInterface ni) {
140                 GridCell c = cellFromCoord(ni.getLocation());
141                 c.addInterface(ni);
142                 ginterfaces.put(ni,c);
143         }
144
```

Disscussion on ConnectivityGrid class and it purpose:

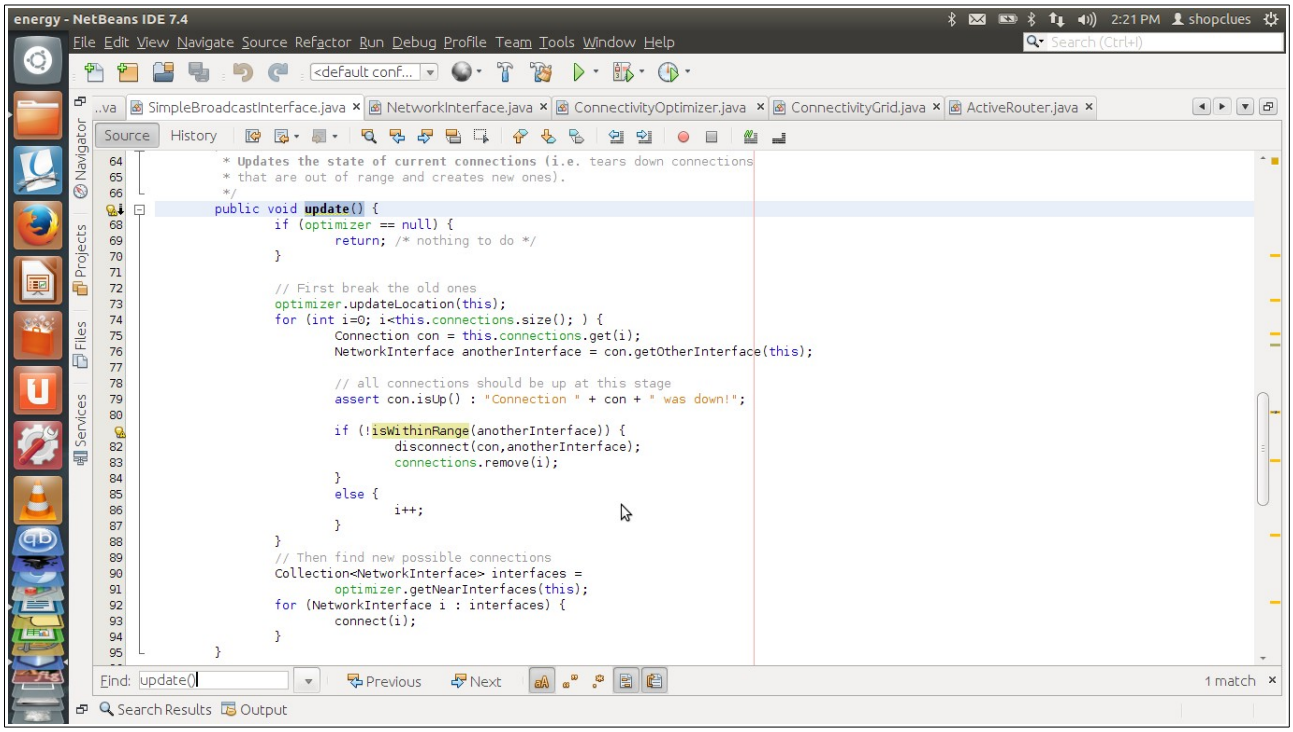


This is the approximate model that describes the functionality of ConnectivityGrid Class.
Here static attribute gridobject associates each ConnectivityGrid to NetworkInterface of Host. This is checked during initialization phase for getting ConnectivityGrid class for particular interface. If not present then one is created with cell array intialised and put into gridobject.
Now according to the location of the node and so the interface, the proper cell from cells array is taken.
The interface is added in the inner class object of GridCell and GridCell object joined the interface in ginterface object. This ConnectivityGrid is also taken in attribute of NetworkInterface Class in the name of optimizer. This also explains why two different interface can't communicate even if they are provided the same connection parameters(transmission speed,range etc).
Since getNeighborCellsByCoord method will only return interface of nodes having same interface class like SimpleBroadcastInterface.
This information will be helpful when we come back to the update method of SimpleBroadcastInterface update method. Understanding of making and breaking connection will become much clear.

Moving back to update method of DTNhost from where this all intermediate story started.

```
328
329         /**
330          * Updates node's network layer and router.
331          * @param simulateConnections Should network layer be updated too
332          */
            public void update(boolean simulateConnections) {
334             if (!isRadioActive()) {
335                     // Make sure inactive nodes don't have connections
336                     tearDownAllConnections();
337                     return;
338             }
339
340             if (simulateConnections) {
341                     for (NetworkInterface i : net) {
342                             i.update();
343                     }
344             }
345             this.router.update();
346     }
```

Now its easy to analyse what this update method is doing.



Let analyse code in three parts.
First:
optimizer.updateLocation(this);

Second:

```
for (int i=0; i<this.connections.size(); ) {
                    Connection con = this.connections.get(i);
                    NetworkInterface anotherInterface = con.getOtherInterface(this);

                    // all connections should be up at this stage
                    assert con.isUp() : "Connection " + con + " was down!";

                    if (!isWithinRange(anotherInterface)) {
                            disconnect(con,anotherInterface);
                            connections.remove(i);
```

```
        }
        else {
                i++;
        }
}
```

Third:

```
// Then find new possible connections
Collection<NetworkInterface> interfaces =
        optimizer.getNearInterfaces(this);
for (NetworkInterface i : interfaces) {
        connect(i);
}
```

First setion just update the cell it must belong to based on it current location.

```
166
167         /**
168          * Checks and updates (if necessary) interface's position in the grid
169          * @param ni The interface to update
170          */
171         public void updateLocation(NetworkInterface ni) {
172                 GridCell oldCell = (GridCell)ginterfaces.get(ni);
173                 GridCell newCell = cellFromCoord(ni.getLocation());
174
175                 if (newCell != oldCell) {
176                         oldCell.moveInterface(ni, newCell);
177                         ginterfaces.put(ni,newCell);
178                 }
179         }
180
```

```
207
208         /**
209          * Get the cell having the specific coordinates
210          * @param c Coordinates
211          * @return The cell
212          */
213         private GridCell cellFromCoord(Coord c) {
214                 // +1 due empty cells on both sides of the matrix
215                 int row = (int)(c.getY()/cellSize) + 1;
216                 int col = (int)(c.getX()/cellSize) + 1;
217
218                 assert row > 0 && row <= rows && col > 0 && col <= cols : "Location " +
219                 c + " is out of world's bounds";
220
221                 return this.cells[row][col];
222         }
223
```

Now second section is easy. It removes those connection which are not in range. Here some interesting thing is going in dissconnect method and same in connect of third section.

```
340
341         /**
342          * Returns true if another interface is within radio range of this interface
343          * and this interface is also within radio range of the another interface.
344          * @param anotherInterface The another interface
345          * @return True if the interface is within range, false if not
346          */
347         protected boolean isWithinRange(NetworkInterface anotherInterface) {
348                 double smallerRange = anotherInterface.getTransmitRange();
349                 double myRange = getTransmitRange();
350                 if (myRange < smallerRange) {
351                         smallerRange = myRange;
352                 }
353
354                 return this.host.getLocation().distance(
355                                 anotherInterface.getHost().getLocation()) <= smallerRange;
356         }
357
```

```
320
321         /**
322          * Disconnects this host from another host.  The derived class should
323          * make the decision whether to disconnect or not
324          * @param con The connection to tear down
325          */
326         protected void disconnect(Connection con,
                        NetworkInterface anotherInterface) {
328                 con.setUpState(false);
329                 notifyConnectionListeners(CON_DOWN, anotherInterface.getHost());
330
331                 // tear down bidirectional connection
332                 if (!anotherInterface.getConnections().remove(con)) {
333                         throw new SimError("No connection " + con + " found in " +
334                                         anotherInterface);
335                 }
336
337                 this.host.connectionDown(con);
338                 anotherInterface.getHost().connectionDown(con);
339         }
340
```

Here connectionDown method of DTNHost is called. Let us see what happened there.

```
    /**
         * Informs the router of this host about state change in a connection
         * object.
         * @param con  The connection object whose state changed
         */
        public void connectionUp(Connection con) {
                this.router.changedConnection(con);
        }

        public void connectionDown(Connection con) {
                this.router.changedConnection(con);
        }
```

Important point is *this.router.changedConnection(con)* is called from here. ProphetRouter uses this function to update it delivery probabilty value. Also *con.setUpState(false)* is function make this.isUp = false which is again used in Prophet Router.

Now Third section First it gets near interface. Near interface are those who lie in +- 1 row and column in cells array.

```
        /**
         * Returns all interfaces that are "near" (i.e., in neighboring grid cells)
         * and use the same technology and channel as the given interface
         * @param ni The interface whose neighboring interfaces are returned
         * @return List of near interfaces
         */
        public Collection<NetworkInterface> getNearInterfaces(
                        NetworkInterface ni) {
                ArrayList<NetworkInterface> niList = new ArrayList<NetworkInterface>();
                GridCell loc = (GridCell)ginterfaces.get(ni);

                if (loc != null) {
                        GridCell[] neighbors =
                                getNeighborCellsByCoord(ni.getLocation());
                        for (int i=0; i < neighbors.length; i++) {
                                niList.addAll(neighbors[i].getInterfaces());
                        }
                }

                return niList;
        }
```

```
        /**
         * Finds all neighboring cells and the cell itself based on the coordinates
         * @param c The coordinates
         * @return Array of neighboring cells
         */
        private GridCell[] getNeighborCellsByCoord(Coord c) {
                // +1 due empty cells on both sides of the matrix
                int row = (int)(c.getY()/cellSize) + 1;
                int col = (int)(c.getX()/cellSize) + 1;
                return getNeighborCells(row,col);
        }
```

```
        /**
         * Returns an array of Cells that contains the neighbors of a certain
         * cell and the cell itself.
         * @param row Row index of the cell
         * @param col Column index of the cell
         * @return Array of neighboring Cells
         */
        private GridCell[] getNeighborCells(int row, int col) {
                return new GridCell[] {
                        cells[row-1][col-1],cells[row-1][col],cells[row-1][col+1],//1st row
                        cells[row][col-1],cells[row][col],cells[row][col+1],//2nd row
                        cells[row+1][col-1],cells[row+1][col],cells[row+1][col+1]//3rd row
                };
        }
```

Now back to update Method of SimpleBroadcastInterface. Here connect Method checks the returned nearby interfaces which are in range or not. Then make new Connection.

Connection con = new CBRConnection(this.host, this,anotherInterface.getHost(), anotherInterface, conSpeed);

Then connect Method is called which is again *this.host.connectionUp(con).*

```
        public void connect(NetworkInterface anotherInterface) {
                if (isScanning()
                                && anotherInterface.getHost().isRadioActive()
                                && isWithinRange(anotherInterface)
                                && !isConnected(anotherInterface)
                                && (this != anotherInterface)) {
                        // new contact within range
                        // connection speed is the lower one of the two speeds
                        int conSpeed = anotherInterface.getTransmitSpeed();
                        if (conSpeed > this.transmitSpeed) {
                                conSpeed = this.transmitSpeed;
                        }

                        Connection con = new CBRConnection(this.host, this,
                                        anotherInterface.getHost(), anotherInterface, conSpeed);
                        connect(con,anotherInterface);
                }
        }
```

```
        /**
         * Connects this host to another host. The derived class should check
         * that all pre-requisites for making a connection are satisfied before
         * actually connecting.
         * @param con The new connection object
         * @param anotherInterface The interface to connect to
         */
        protected void connect(Connection con, NetworkInterface anotherInterface) {
                this.connections.add(con);
                notifyConnectionListeners(CON_UP, anotherInterface.getHost());

                // set up bidirectional connection
                anotherInterface.getConnections().add(con);

                // inform routers about the connection
                this.host.connectionUp(con);
                anotherInterface.getHost().connectionUp(con);
        }
```

Here Note that Bidirectional Connections are made. So if any body wants to have one way connection, they can dissable or comment any *add* method based on their conditions.

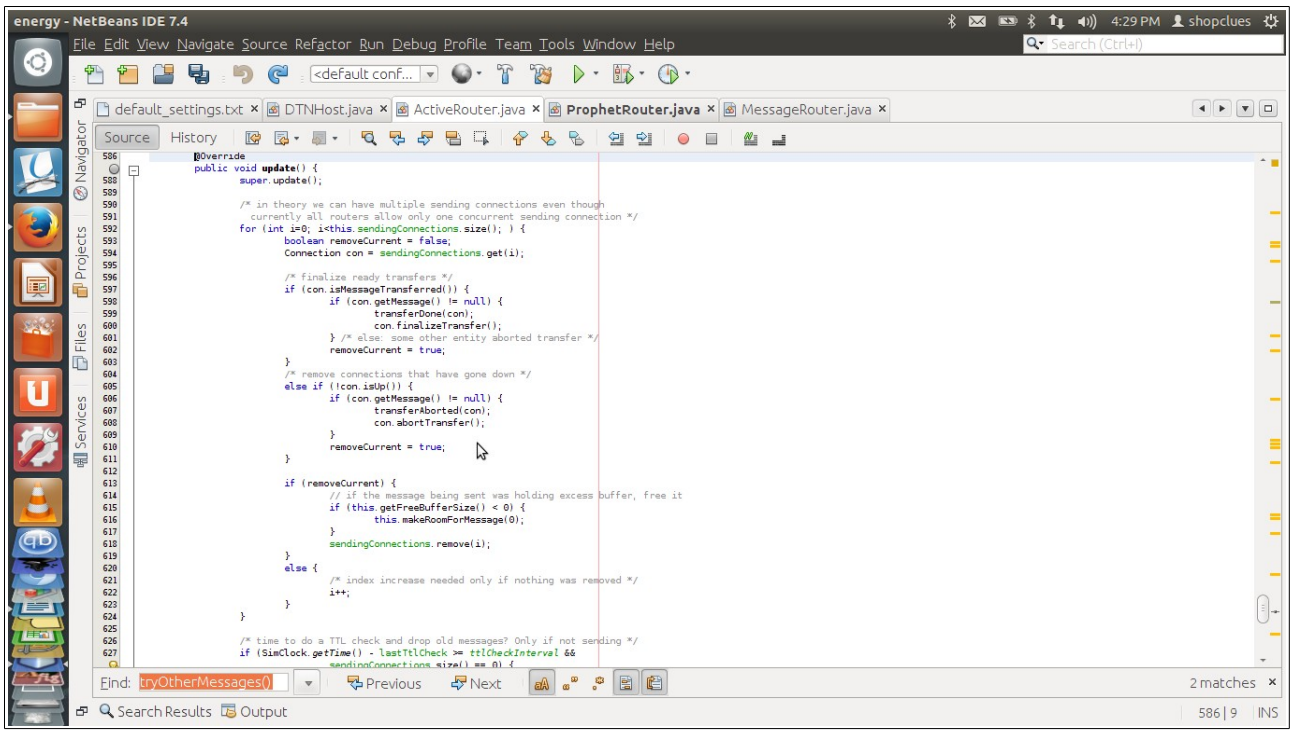
```
328
329         /**
330          * Updates node's network layer and router.
331          * @param simulateConnections Should network layer be updated too
332          */
333         public void update(boolean simulateConnections) {
334                 if (!isRadioActive()) {
335                         // Make sure inactive nodes don't have connections
336                         tearDownAllConnections();
337                         return;
338                 }
339
340                 if (simulateConnections) {
341                         for (NetworkInterface i : net) {
342                                 i.update();
343                         }
344                 }
345                 this.router.update();
346         }
```

Now back to *this.router.update()* method. It reaches the to ProphetRouter update Method in our case study scenario.

```
        @Override
        public void update() {

                super.update();
                if (!canStartTransfer() ||isTransferring()) {
                        return; // nothing to transfer or is currently transferring
                }

                // try messages that could be delivered to final recipient
                if (exchangeDeliverableMessages() != null) {
                        return;
                }

                tryOtherMessages();
        }
```

Here we observe Five calls:
1. super.update()
2. canStartTransfer()
3. isTransferring()
4. exchangeDeliverableMessages()
5. tryOtherMessages()

Lets us discuss all activities of routing one by one.




```java
/**
 * Updates router.
 * This method should be called (at least once) on every simulation
 * interval to update the status of transfer(s).
 */
public void update(){
	for (Collection<Application> apps : this.applications.values()) {
		for (Application app : apps) {
			app.update(this.host);
		}
	}
}
```

First it enters the update of ActiveRouter class. *super.update()* send control to MessageRouter update Method which updates application layer. In our case study it is not used.

Comming back to ActiveRouter.update Method. Let us see for loop iteration. SendingConnection contains the list of connection which has started transfering the Message.

```java
/**
 * Returns true if the current message transfer is done.
 * @return True if the transfer is done, false if not
 */
public boolean isMessageTransferred() {
	return getRemainingByteCount() == 0;
}
```

```
        /**
         * Returns the amount of bytes to be transferred before ongoing transfer
         * is ready or 0 if there's no ongoing transfer or it has finished
         * already
         * @return the amount of bytes to be transferred
         */
        public int getRemainingByteCount() {
                int remaining;

                if (msgOnFly == null) {
                        return 0;
                }

                remaining = (int)((this.transferDoneTime - SimClock.getTime())
                                * this.speed);

                return (remaining > 0 ? remaining : 0);
        }
```

This method checks whether message has been transferred or not. this.transferDoneTime is calculated as per the message size which has been send through this connection. Next *con.getMessage()* returns msgOnFly which tells that message is finalised or not. All these methods will going to be discussed in the Routing part discussion.

```
        /**
         * Gets the message that this connection is currently transferring.
         * @return The message or null if no message is being transferred
         */
        public Message getMessage() {
                return this.msgOnFly;
        }
```

Then *transferDone(con)* method is doing nothing here.

```
        public void finalizeTransfer() {
                assert this.msgOnFly != null : "Nothing to finalize in " + this;
                assert msgFromNode != null : "msgFromNode is not set";

                this.bytesTransferred += msgOnFly.getSize();

                getOtherNode(msgFromNode).messageTransferred(this.msgOnFly.getId(),
                                msgFromNode);
                clearMsgOnFly();
        }
```

The *con.abortTransfer()*

```
        /**
         * Clears the message that is currently being transferred.
         * Calls to {@link #getMessage()} will return null after this.
         */
        protected void clearMsgOnFly() {
                this.msgOnFly = null;
                this.msgFromNode = null;
        }
```

The *con.isUp()* is used which is controlled in NetworkInterface update method.

```
        public void abortTransfer() {
                assert msgOnFly != null : "No message to abort at " + msgFromNode;
                int bytesRemaining = getRemainingByteCount();

                this.bytesTransferred += msgOnFly.getSize() - bytesRemaining;

                getOtherNode(msgFromNode).messageAborted(this.msgOnFly.getId(),
                                msgFromNode, bytesRemaining);
                clearMsgOnFly();
        }
```

```
482
483         /**
484          * Informs the host that a message transfer was aborted.
485          * @param id Identifier of the message
486          * @param from From who the message was from
487          * @param bytesRemaining Nrof bytes that were left before the transfer
488          * would have been ready; or -1 if the number of bytes is not known
489          */
            public void messageAborted(String id, DTNHost from, int bytesRemaining) {
491             this.router.messageAborted(id, from, bytesRemaining);
492         }
493
```

```
458
459         /**
460          * This method should be called (on the receiving host) when a message
461          * transfer was aborted.
462          * @param id Id of the message that was being transferred
463          * @param from Host the message was from (previous hop)
464          * @param bytesRemaining Nrof bytes that were left before the transfer
465          * would have been ready; or -1 if the number of bytes is not known
466          */
            public void messageAborted(String id, DTNHost from, int bytesRemaining) {
468             Message incoming = removeFromIncomingBuffer(id, from);
469             if (incoming == null) {
470                     throw new SimError("No incoming message for id " + id +
471                                     " to abort in " + this.host);
472             }
473
474             for (MessageListener ml : this.mListeners) {
475                     ml.messageTransferAborted(incoming, from, this.host);
476             }
477         }
```

In rest part, if removeCurrent is true then it makes space in message buffer if required and removes sendingconnections. Otherwise increment *i* which means move to next connection to check.
If TTL is expired and it is not transferring the message then messages are dropped whose ttl is expired.

```
295
296         /**
297          * Drops messages whose TTL is less than zero.
298          */
299         protected void dropExpiredMessages() {
300                 Message[] messages = getMessageCollection().toArray(new Message[0]);
                    for (int i=0; i<messages.length; i++) {
302                         int ttl = messages[i].getTtl();
303                         if (ttl <= 0) {
304                                 deleteMessage(messages[i].getId(), true);
305                         }
306                 }
307         }
308
```

```
490
491         /**
492          * Deletes a message from the buffer and informs message listeners
493          * about the event
494          * @param id Identifier of the message to delete
495          * @param drop If the message is dropped (e.g. because of full buffer) this
496          * should be set to true. False value indicates e.g. remove of message
497          * because it was delivered to final destination.
498          */
            public void deleteMessage(String id, boolean drop) {
500             Message removed = removeFromMessages(id);
501             if (removed == null) throw new SimError("no message for id " +
502                             id + " to remove at " + this.host);
503
504             for (MessageListener ml : this.mListeners) {
505                     ml.messageDeleted(removed, this.host, drop);
506             }
507         }
508
```

Also at last if energy is defined it updates its value also.

```
                if (energy != null) {
                        /* TODO: add support for other interfaces */
                        NetworkInterface iface = getHost().getInterface(1);
                        energy.update(iface, getHost().getComBus());
                }
```

Now comming back to *update* Method of Prophet.

```
        /**
         * Makes rudimentary checks (that we have at least one message and one
         * connection) about can this router start transfer.
         * @return True if router can start transfer, false if not
         */
        protected boolean canStartTransfer() {
                if (this.getNrofMessages() == 0) {
                        return false;
                }
                if (this.getConnections().size() == 0) {
                        return false;
                }

                return true;
        }
```

It basically checks either number of messages in message buffer is not zero or connection list must not be empty.

```
        /**
         * Returns true if this router is transferring something at the moment or
         * some transfer has not been finalized.
         * @return true if this router is transferring something
         */
        public boolean isTransferring() {
                if (this.sendingConnections.size() > 0) {
                        return true; // sending something
                }

                List<Connection> connections = getConnections();

                if (connections.size() == 0) {
                        return false; // not connected
                }

                for (int i=0, n=connections.size(); i<n; i++) {
                        Connection con = connections.get(i);
                        if (!con.isReadyForTransfer()) {
                                return true;    // a connection isn't ready for new transfer
                        }
                }

                return false;
        }
```

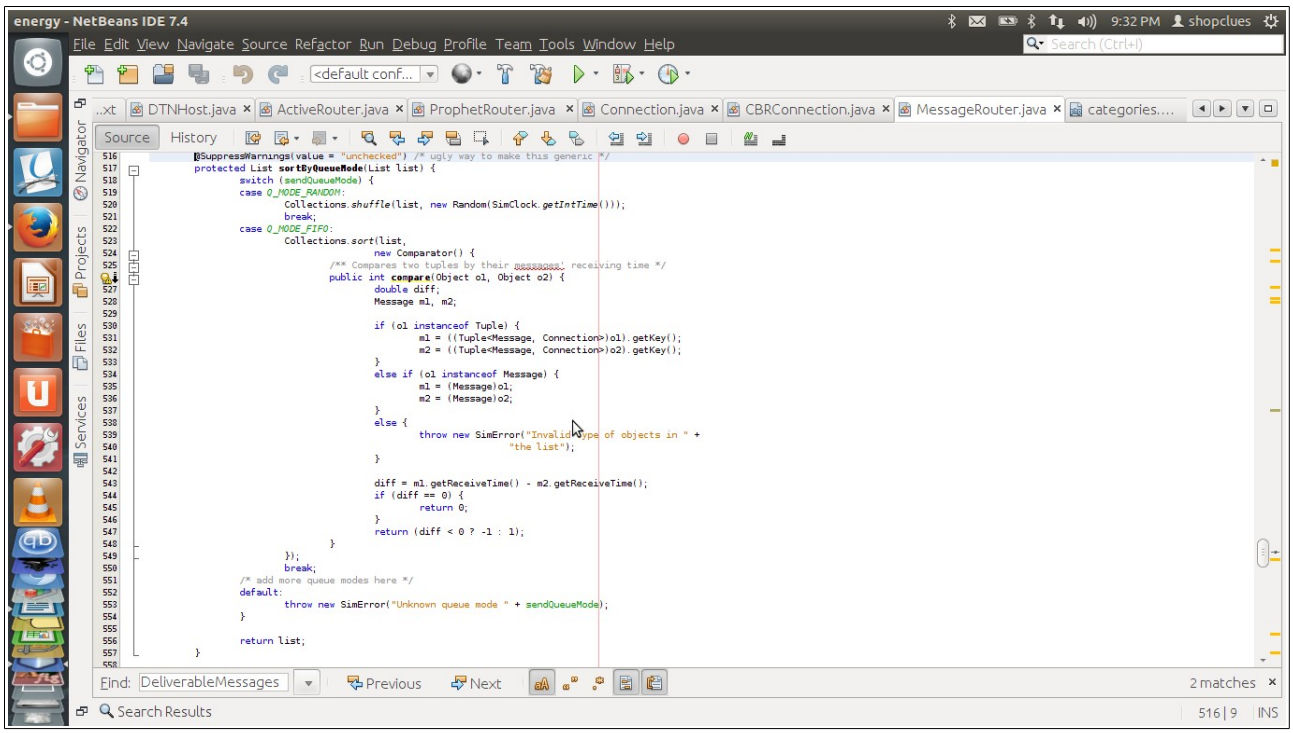
```
        /**
         * Returns true if the connection is ready to transfer a message (connection
         * is up and there is no message being transferred).
         * @return true if the connection is ready to transfer a message
         */
        public boolean isReadyForTransfer() {
                return this.isUp && this.msgOnFly == null;
        }
```

IsTransferring() method checks whether connection are made or not.

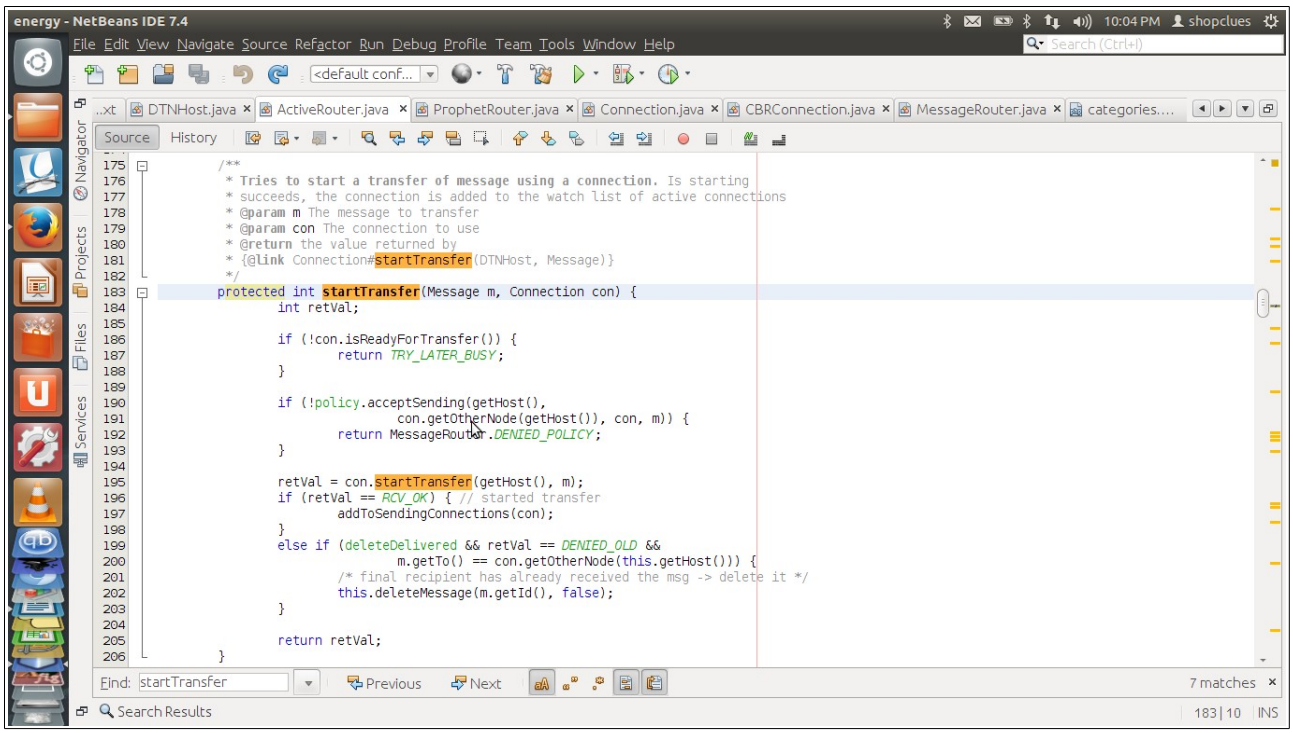
```
468
469         /**
470          * Exchanges deliverable (to final recipient) messages between this host
471          * and all hosts this host is currently connected to. First all messages
472          * from this host are checked and then all other hosts are asked for
473          * messages to this host. If a transfer is started, the search ends.
474          * @return A connection that started a transfer or null if no transfer
475          * was started
476          */
477         protected Connection exchangeDeliverableMessages() {
478                 List<Connection> connections = getConnections();
479
480                 if (connections.size() == 0) {
481                         return null;
482                 }
483
484                 @SuppressWarnings(value = "unchecked")
485                 Tuple<Message, Connection> t =
486                         tryMessagesForConnected(sortByQueueMode(getMessagesForConnected()));
487
488                 if (t != null) {
489                         return t.getValue(); // started transfer
490                 }
491
492                 // didn't start transfer to any node -> ask messages from connected
493                 for (Connection con : connections) {
494                         if (con.getOtherNode(getHost()).requestDeliverableMessages(con)) {
495                                 return con;
496                         }
497                 }
498
499                 return null;
500         }
501
```

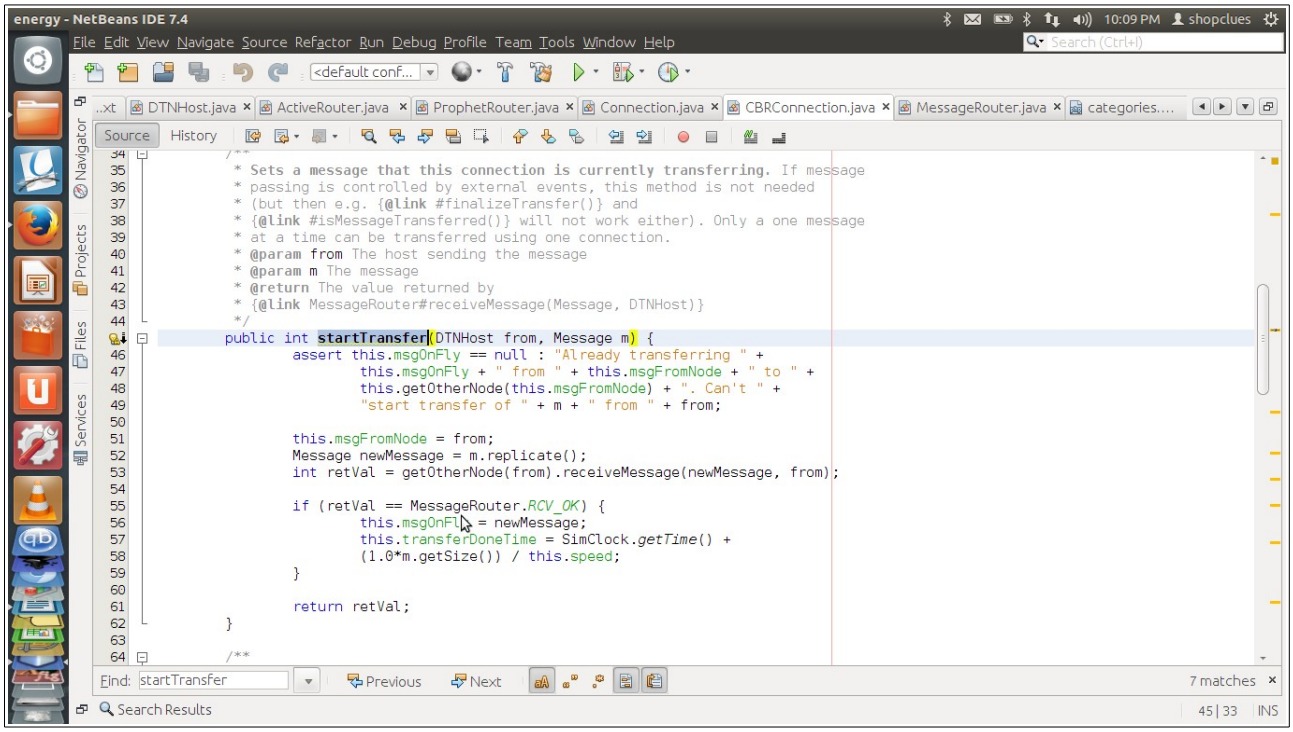
```
351         /**
352          * Returns a list of message-connections tuples of the messages whose
353          * recipient is some host that we're connected to at the moment.
354          * @return a list of message-connections tuples
355          */
356         protected List<Tuple<Message, Connection>> getMessagesForConnected() {
357                 if (getNrofMessages() == 0 || getConnections().size() == 0) {
358                         /* no messages -> empty list */
359                         return new ArrayList<Tuple<Message, Connection>>(0);
360                 }
361
362                 List<Tuple<Message, Connection>> forTuples =
363                         new ArrayList<Tuple<Message, Connection>>();
364                 for (Message m : getMessageCollection()) {
365                         for (Connection con : getConnections()) {
366                                 DTNHost to = con.getOtherNode(getHost());
367                                 if (m.getTo() == to) {
368                                         forTuples.add(new Tuple<Message, Connection>(m,con));
369                                 }
370                         }
371                 }
372
373                 return forTuples;
374         }
375
```

redundant type arguments in new expression (use diamond operat
---
(Alt-Enter shows hints)




```
376         /**
377          * Tries to send messages for the connections that are mentioned
378          * in the Tuples in the order they are in the list until one of
379          * the connections starts transferring or all tuples have been tried.
380          * @param tuples The tuples to try
381          * @return The tuple whose connection accepted the message or null if
382          * none of the connections accepted the message that was meant for them.
383          */
384         protected Tuple<Message, Connection> tryMessagesForConnected(
385                         List<Tuple<Message, Connection>> tuples) {
386                 if (tuples.size() == 0) {
387                         return null;
388                 }
389
390                 for (Tuple<Message, Connection> t : tuples) {
391                         Message m = t.getKey();
392                         Connection con = t.getValue();
393                         if (startTransfer(m, con) == RCV_OK) {
394                                 return t;
395                         }
396                 }
397
398                 return null;
399         }
```

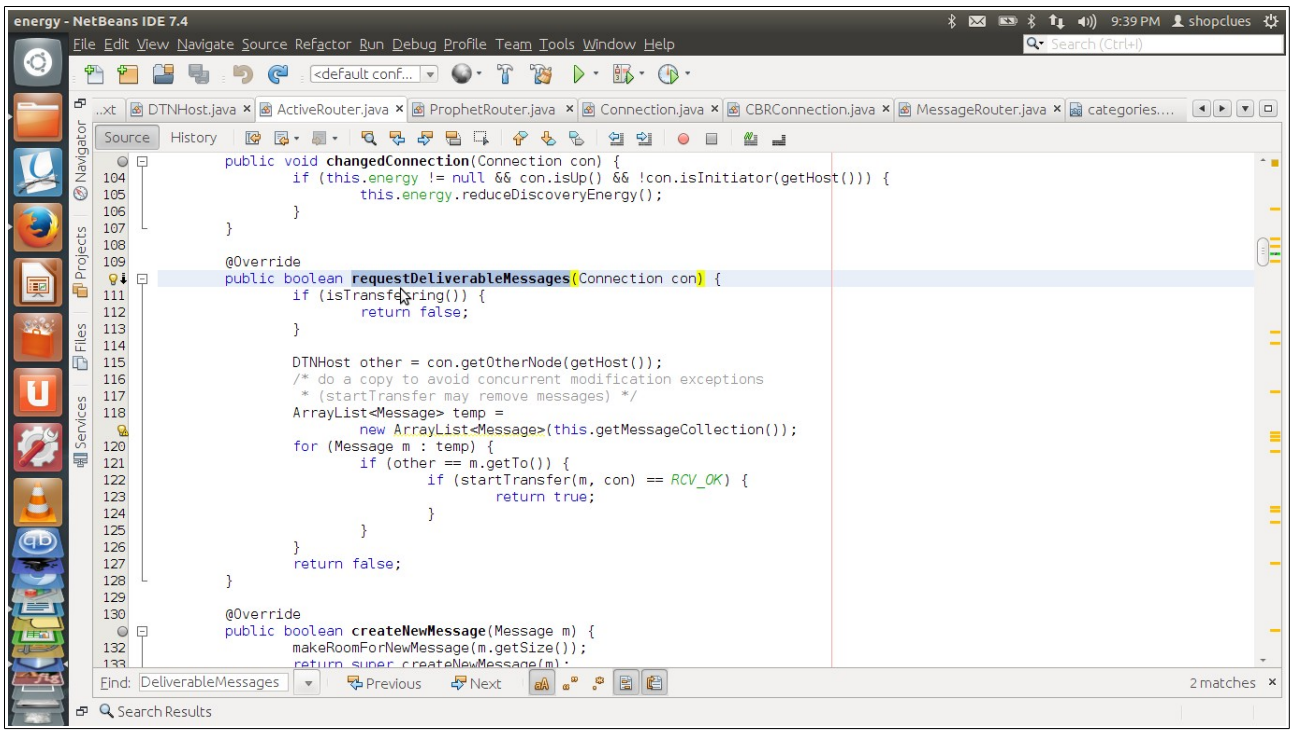

energy - NetBeans IDE 7.4
10:04 PM shopclues
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
DTNHost.java
ActiveRouter.java
ProphetRouter.java
Connection.java
CBRConnection.java
MessageRouter.java
categories....
Source
History
/**
* Tries to start a transfer of message using a connection. Is starting
* succeeds, the connection is added to the watch list of active connections
* @param m The message to transfer
* @param con The connection to use
* @return the value returned by
* {@link Connection#startTransfer(DTNHost, Message)}
*/
protected int startTransfer(Message m, Connection con) {
int retVal;
if (!con.isReadyForTransfer()) {
return TRY_LATER_BUSY;
}
if (!policy.acceptSending(getHost(),
con.getOtherNode(getHost()), con, m)) {
return MessageRouter.DENIED_POLICY;
}
retVal = con.startTransfer(getHost(), m);
if (retVal == RCV_OK) { // started transfer
addToSendingConnections(con);
}
else if (deleteDelivered && retVal == DENIED_OLD &&
m.getTo() == con.getOtherNode(this.getHost())) {
/* final recipient has already received the msg -> delete it */
this.deleteMessage(m.getId(), false);
}
return retVal;
}
Find: startTransfer
Previous
Next
7 matches
Search Results
183 | 10
INS


energy - NetBeans IDE 7.4
10:09 PM shopclues
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
DTNHost.java
ActiveRouter.java
ProphetRouter.java
Connection.java
CBRConnection.java
MessageRouter.java
categories....
Source
History
/**
* Sets a message that this connection is currently transferring. If message
* passing is controlled by external events, this method is not needed
* (but then e.g. {@link #finalizeTransfer()} and
* {@link #isMessageTransferred()} will not work either). Only a one message
* at a time can be transferred using one connection.
* @param from The host sending the message
* @param m The message
* @return The value returned by
* {@link MessageRouter#receiveMessage(Message, DTNHost)}
*/
public int startTransfer(DTNHost from, Message m) {
assert this.msgOnFly == null : "Already transferring " +
this.msgOnFly + " from " + this.msgFromNode + " to " +
this.getOtherNode(this.msgFromNode) + ". Can't " +
"start transfer of " + m + " from " + from;
this.msgFromNode = from;
Message newMessage = m.replicate();
int retVal = getOtherNode(from).receiveMessage(newMessage, from);
if (retVal == MessageRouter.RCV_OK) {
this.msgOnFly = newMessage;
this.transferDoneTime = SimClock.getTime() +
(1.0*m.getSize()) / this.speed;
}
return retVal;
}
Find: startTransfer
Previous
Next
7 matches
Search Results
45 | 33
INS

```java
446
447         /**
448          * Start receiving a message from another host
449          * @param m The message
450          * @param from Who the message is from
451          * @return The value returned by
452          * {@link MessageRouter#receiveMessage(Message, DTNHost)}
453          */
454         public int receiveMessage(Message m, DTNHost from) {
455                 int retVal = this.router.receiveMessage(m, from);
456
457                 if (retVal == MessageRouter.RCV_OK) {
458                         m.addNodeOnPath(this);  // add this node on the messages path
459                 }
460
461                 return retVal;
462         }
```

```java
        @Override
        public int receiveMessage(Message m, DTNHost from) {
                int recvCheck = checkReceiving(m, from);
                if (recvCheck != RCV_OK) {
                        return recvCheck;
                }

                // seems OK, start receiving the message
                return super.receiveMessage(m, from);
        }
```

```java
        /**
         * Try to start receiving a message from another host.
         * @param m Message to put in the receiving buffer
         * @param from Who the message is from
         * @return Value zero if the node accepted the message (RCV_OK), value less
         * than zero if node rejected the message (e.g. DENIED_OLD), value bigger
         * than zero if the other node should try later (e.g. TRY_LATER_BUSY).
         */
        public int receiveMessage(Message m, DTNHost from) {
                Message newMessage = m.replicate();

                this.putToIncomingBuffer(newMessage, from);
                newMessage.addNodeOnPath(this.host);

                for (MessageListener ml : this.mListeners) {
                        ml.messageTransferStarted(newMessage, from, getHost());
                }

                return RCV_OK; // superclass always accepts messages
        }
```



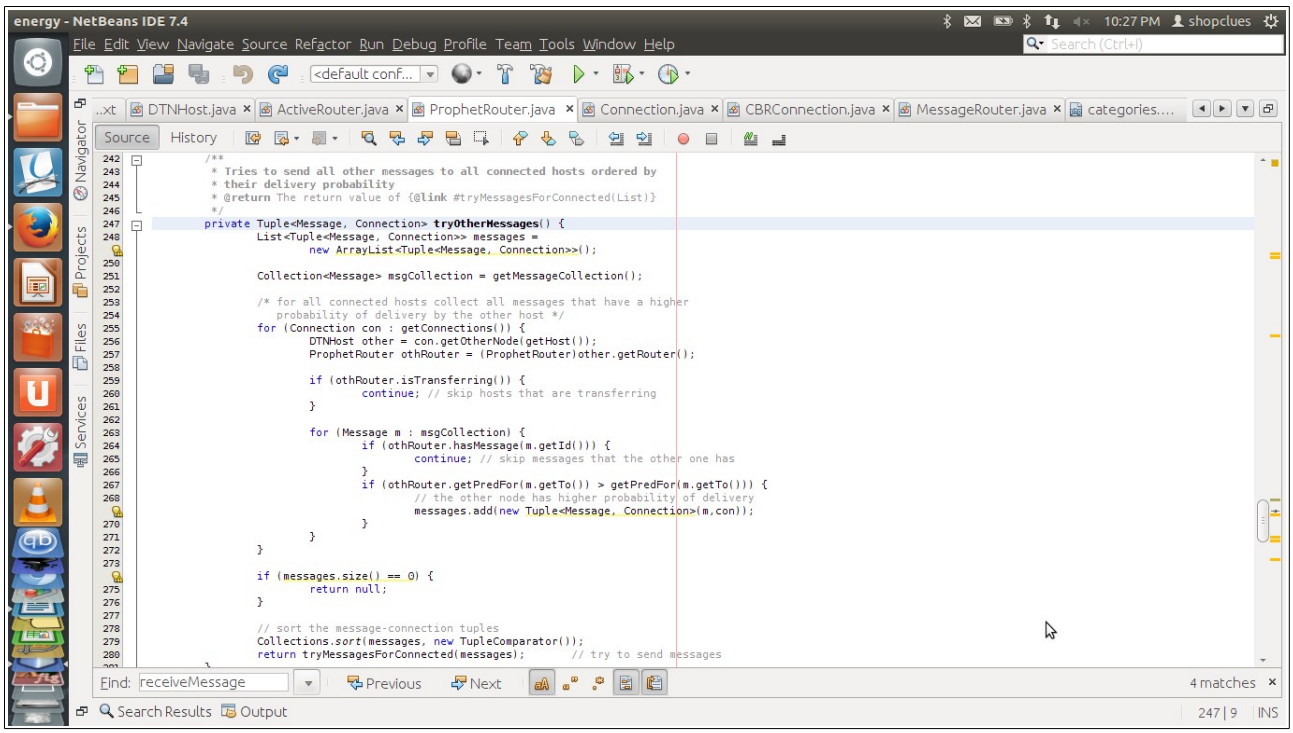
exchangeDeliverableMessages() Method starts transfer of message between two nodes. Message is excanged only if other node is destination. Let us see what happens one by one.
RequestDeliverableMessages. We observe four major method here.

1. getMessagesForConnected
2. sortByQueueMode
3. tryMessagesForConnected
4. requestDeliverableMessages

First Method returns tuple object with message and connection list. The host associated with connection must be destination to message. In second method, messages are suffle from list on bases of two modes taken from default config file. Two modes are FIFO and RAMDOM modes. Third starts transfer between two nodes. Fourth will also do same as third.

Here startTransfer of ActiveRouter call CBRConnection startTransfer Method. Here we observe the method calling of *getOtherNode(from).receiveMessage(newMessage, from)* and intialisation of *this.msgOnFly = newMessage, this.transferDoneTime = SimClock.getTime() + (1.0*m.getSize()) / this.speed;*
Message is send to incoming buffer of other host. MsgOnFly is set and *this.transferDoneTime* gets value when transfer has to be finished based upon the size of message.

Now again comming out of exchangedeliverablemessage Method. Last call is done to *tryOtherMessages()* method of ProphetRouter.



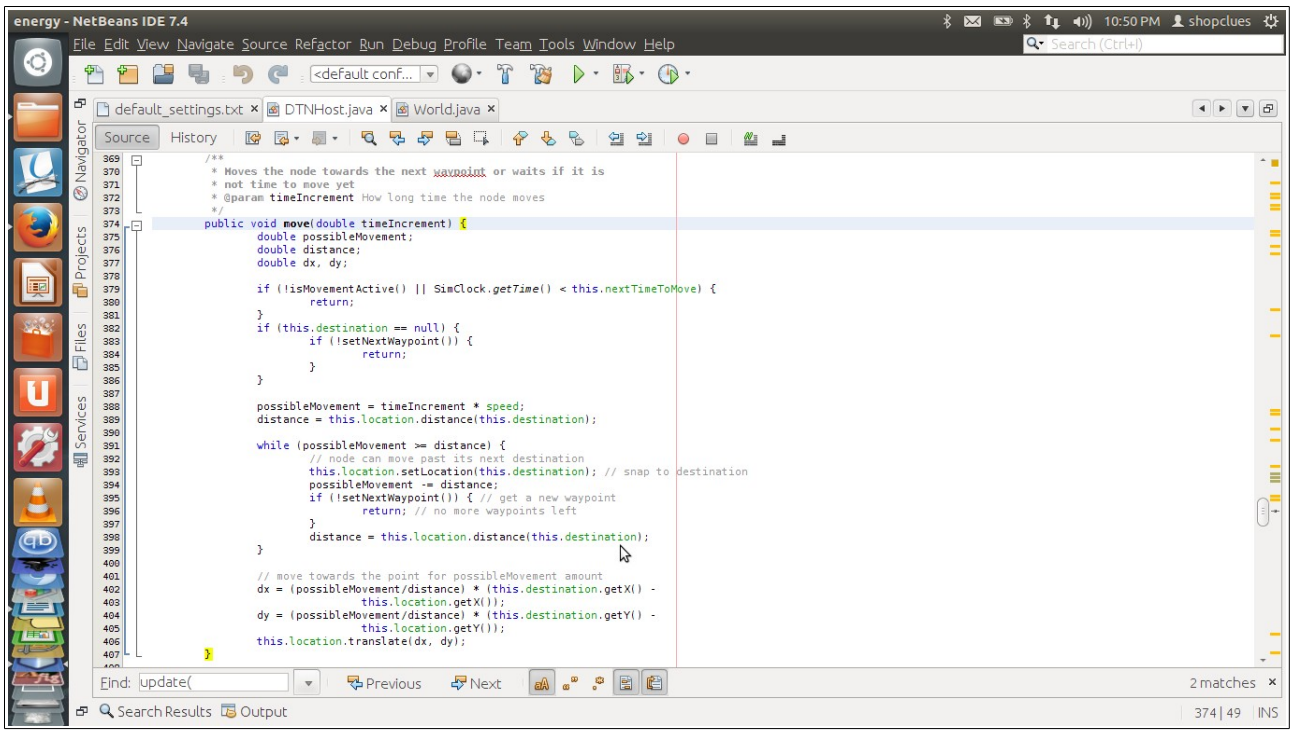
All the code is pretty clear. After all filtering criterias messages list of tuple is created. This is passed to *tryMessagesForConnected(messages).* This method has been already discussed previously.

CBRConnection holds responsibility of the Message transfer information. Connection object made only when interfaces are in range of each other. That define its purpose.

Now back to Original world update Method the *moveHosts(this.updateInterval) which* is yet to discuss.

This method moves the hosts to new location that must be with updateInterval value.

```
        /**
         * Moves all hosts in the world for a given amount of time
         * @param timeIncrement The time how long all nodes should move
         */
        private void moveHosts(double timeIncrement) {
                for (int i=0,n = hosts.size(); i<n; i++) {
                        DTNHost host = hosts.get(i);
                        host.move(timeIncrement);
                }
        }
```




```java
/**
 * Sets the next destination and speed to correspond the next waypoint
 * on the path.
 * @return True if there was a next waypoint to set, false if node still
 * should wait
 */
private boolean setNextWaypoint() {
	if (path == null) {
		path = movement.getPath();
	}

	if (path == null || !path.hasNext()) {
		this.nextTimeToMove = movement.nextPathAvailable();
		this.path = null;
		return false;
	}

	this.destination = path.getNextWaypoint();
	this.speed = path.getSpeed();

	if (this.movListeners != null) {
		for (MovementListener l : this.movListeners) {
			l.newDestination(this, this.destination, this.speed);
		}
	}

	return true;
}
```

```java
	 * Returns a possible (random) placement for a host
	 * @return Random position on the map
	 */
	@Override
	public Coord getInitialLocation() {
		assert rng != null : "MovementModel not initialized!";
		Coord c = randomCoord();

		this.lastWaypoint = c;
		return c;
	}

	@Override
	public Path getPath() {
		Path p;
		p = new Path(generateSpeed());
		p.addWaypoint(lastWaypoint.clone());
		Coord c = lastWaypoint;

		for (int i=0; i<PATH_LENGTH; i++) {
			c = randomCoord();
			p.addWaypoint(c);
		}

		this.lastWaypoint = c;
		return p;
	}
```

```java
	/**
	 * Returns true if the path has more waypoints, false if not
	 * @return true if the path has more waypoints, false if not
	 */
	public boolean hasNext() {
		return nextWpIndex < this.coords.size();
	}
```

```java
	/**
	 * Returns a sim time when the next path is available. This implementation
	 * returns a random time in future that is {@link #WAIT_TIME} from now.
	 * @return The sim time when node should ask the next time for a path
	 */
	public double nextPathAvailable() {
		return SimClock.getTime() + generateWaitTime();
	}
```

```java
	/**
	 * Returns the next waypoint on this path
	 * @return the next waypoint
	 */
	public Coord getNextWaypoint() {
		assert hasNext() : "Path didn't have " + (nextWpIndex+1) + ". waypoint";
		return coords.get(nextWpIndex++);
	}
```

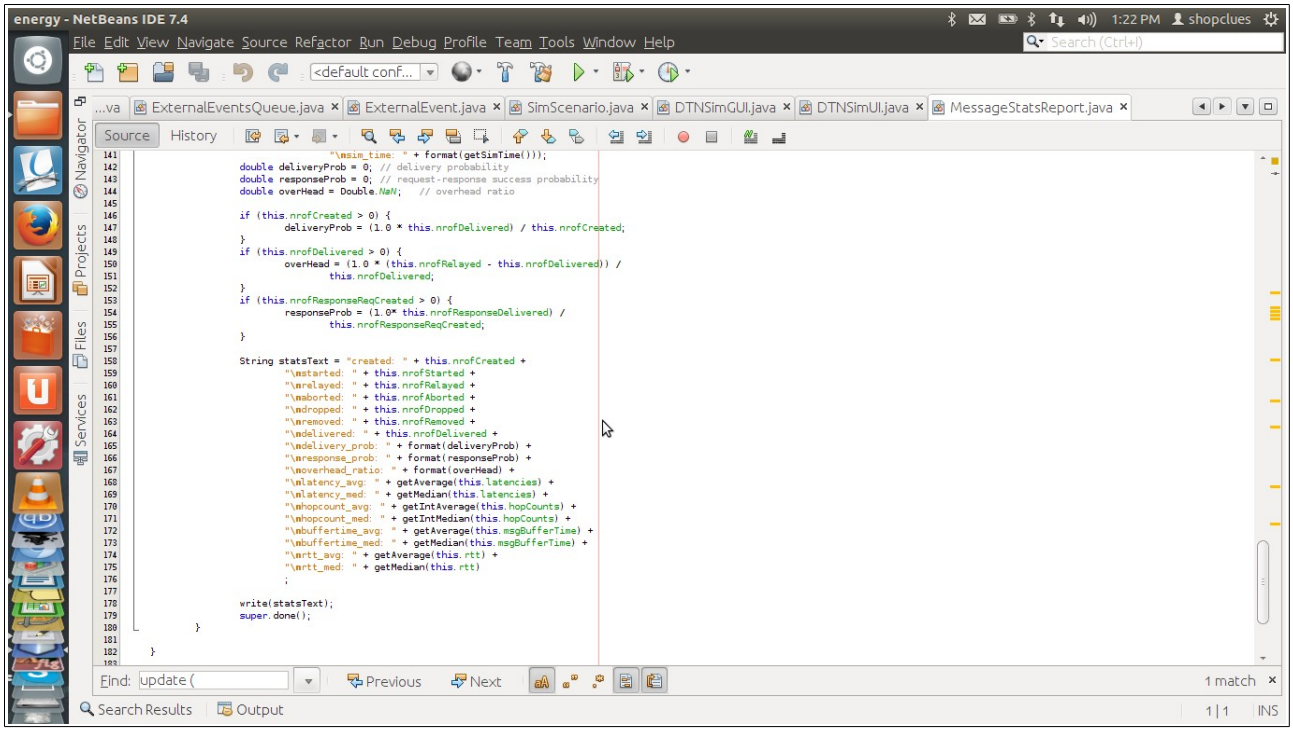
```
        /**
         * Returns the speed towards the next waypoint (asked with
         * {@link #getNextWaypoint()}.
         * @return the speed towards the next waypoint
         */
        public double getSpeed() {
                assert speeds.size() != 0 : "No speed set";
                assert nextWpIndex != 0 : "No waypoint asked";

                if (speeds.size() == 1) {
                        return speeds.get(0);
                }
                else {
                        return speeds.get(nextWpIndex-1);
                }
        }
```

Process starts with basic validation checks. If destination is not set then it obtains its new destination.
This is done trough *setNextWaypoint(). possibleMovement* and *distance* values are compared against each other in the while loop. This while loop makes sure that choosen destination position is not crossed or when node moves to destination position then set host position is same as destination. Then through *setNextWaypoint()* new destination is set. Then distance is calculated in each dimension and position is updated. Lets have a look at *setNextWaypoint()* method. Here first Path is obtained and then destination through *this.destination = path.getNextWaypoint()* and speed is updated and Listeners are informed. In this case scenario it goes to RandomWaypoint getPath Method.

Important discussion on Mobility Model:

During initialisation phase each host gets its location based on Mobility Model. Like in case of MapBasedMobiliy Model each group host will get dispersed on the points in the their WKT file. Reading these file itself clears the concept. Also getPath Method is very important part of Mobilty Model. Since what will be your strategy to choose next destination for node lies here. People interested in modifying mobility according to certain parameters must look this method implementation in various mobility classes. Path class is very simple as it simply carries the coordinates of all places that become its destination point from time to time. It also records the list of speed that it has taken from source to destination. The *nextWpIndex* helps in getting destination point for host as it has lastest index value of List attributes.

At last in DTNSimUI class done() method is called which calls all reports done method included from config.



All the variables of such class is updated through various Method calling of various Listener.
In this case it is MessageListenner.

Possible Network Characterstics in this Research area:

1. Presently only one connection transfer message and bidirectional connection are made.
   We can control this bidirectional property and can make it unidirectional or can control under the condition where A makes connection to B and B to A. It is possible that A is connected to B but
   B is not connected to A.
2. Mobility Pattern on the basis of Message Traffic.
3. Routing Pattern on basis of Mobility Pattern.
4. Queing or Sheduling Policy changes to lighten/reduce the load on Network.
5. Encryption/Decryption of Message.
6. Identity Based Cryptograpy etc.

Once the architecture of ONE is understood ,it is easy to do anything in it. Innovation is the limit.

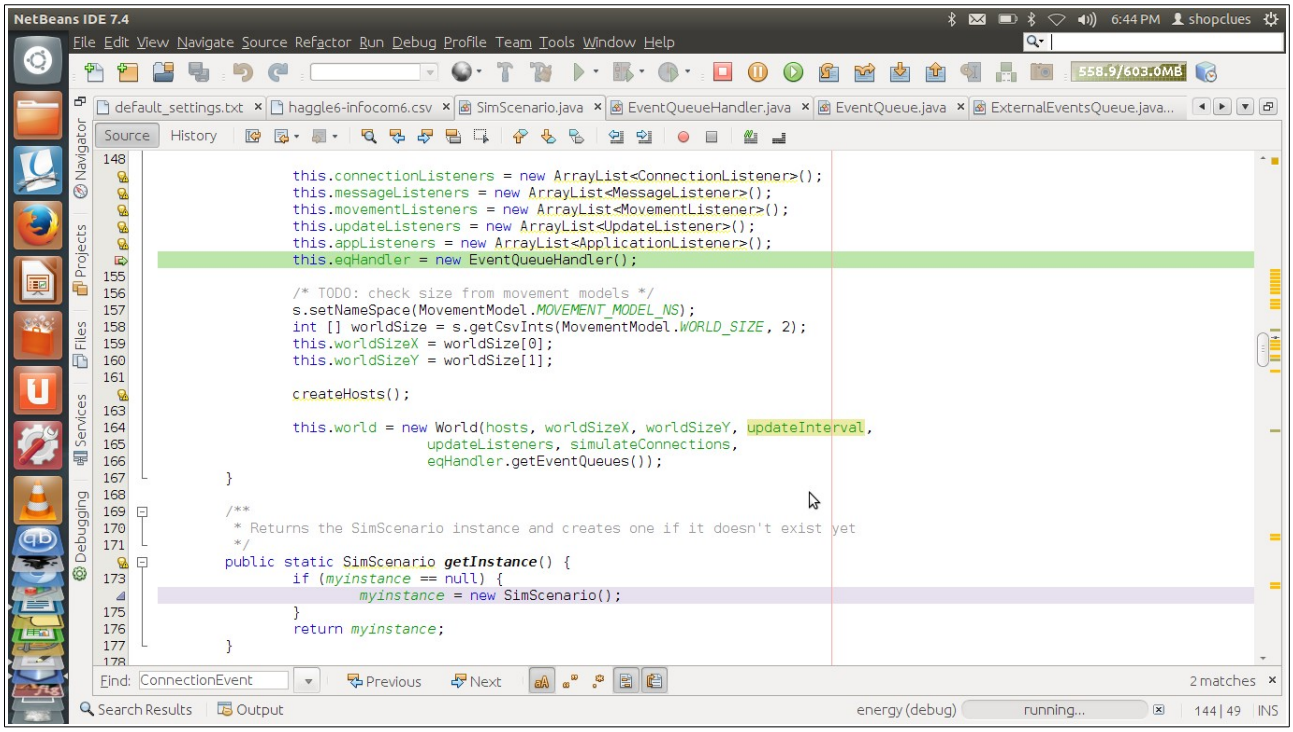
CASE II : External File Analysis in ONE

Introduction

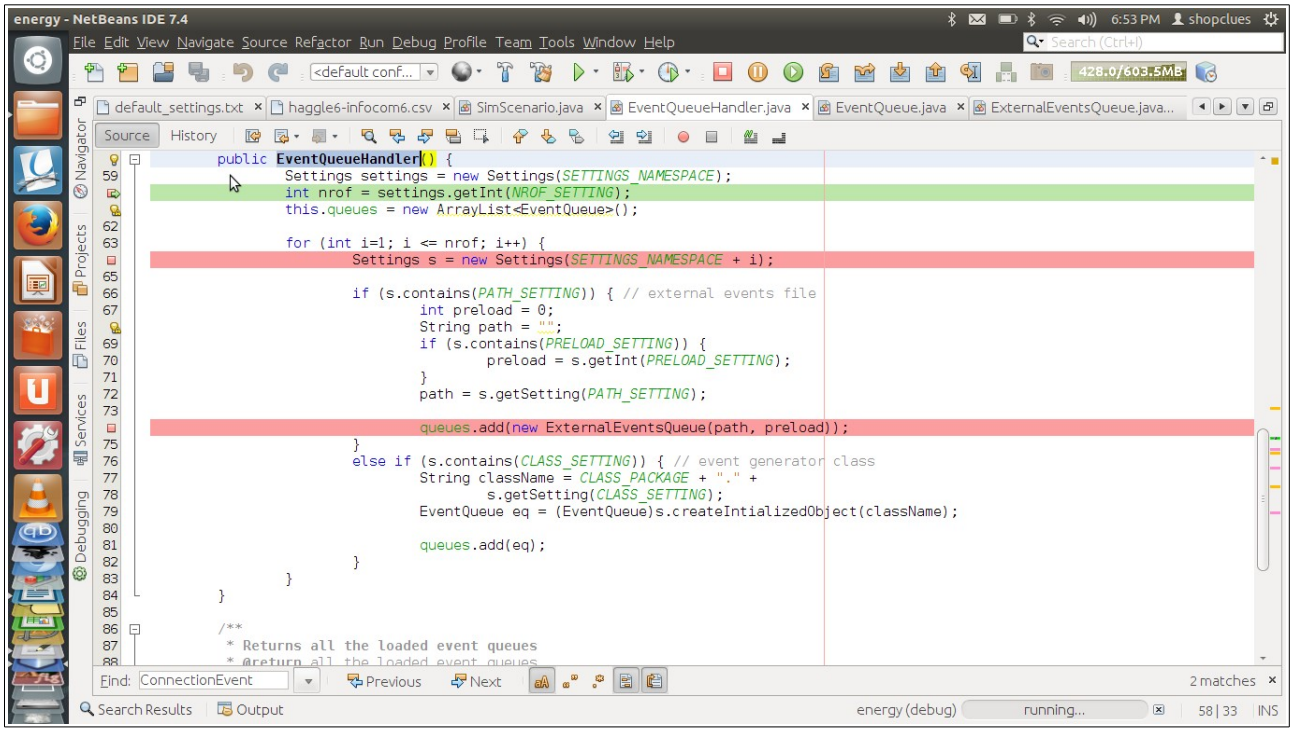
There is also a provision for real data analysis in this simulator. One can define the connection between two nodes and message creation events in specific file format as shown in Appendix section. This section explains the working of code in such scenario.

Initialisation Phase working

Simscenario.java constructure
this.eqHandler = new EventQueueHandler() returns the object that contains the connection up and down timing along with source and destination network address.



EventQueueHandler looks like this:



In this case it goes to *if (s.contains(PATH_SETTING))* part. Here Path is

/home/shopclues/energy/ee/haggle6-infocom6.csv and preload is 0. Then *queues.add(new ExternalEventsQueue(path, preload))* line gets excecuted. In this it reaches to ExternalEventsQueue Constructure.

```
        /**
         * Creates a new Queue from a file
         * @param filePath Path to the file where the events are read from. If
         * file ends with extension defined in {@link BinaryEventsReader#BINARY_EXT}
         * the file is assumed to be a binary file.
         * @param nrofPreload How many events to preload
         * @see BinaryEventsReader#BINARY_EXT
         * @see BinaryEventsReader#storeToBinaryFile(String, List)
         */
        public ExternalEventsQueue(String filePath, int nrofPreload) {
                setNrofPreload(nrofPreload);
                init(filePath);
        }
```

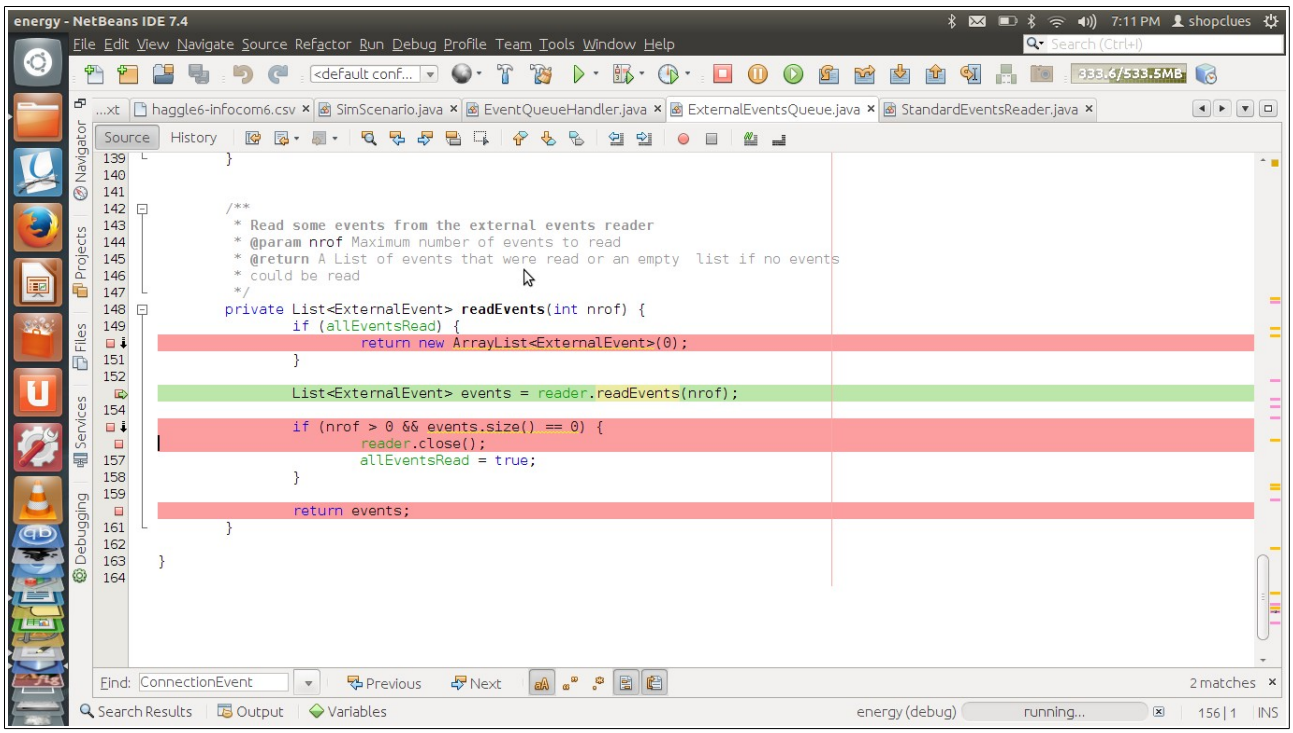
```
        /**
         * Sets maximum number of events that are read when the next preload occurs
         * @param nrof Maximum number of events to read. If less than 1, default
         * value ( {@value DEFAULT_NROF_PRELOAD} ) is used.
         */
        public void setNrofPreload(int nrof) {
                if (nrof < 1) {
                        nrof = DEFAULT_NROF_PRELOAD;
                }
                this.nrofPreload = nrof;
        }
```

```
        private void init(String eeFilePath) {
                this.eventsFile = new File(eeFilePath);

                if (BinaryEventsReader.isBinaryEeFile(eventsFile)) {
                        this.reader = new BinaryEventsReader(eventsFile);
                }
                else {
                        this.reader = new StandardEventsReader(eventsFile);
                }

                this.queue = readEvents(nrofPreload);
                this.nextEventIndex = 0;
        }
```

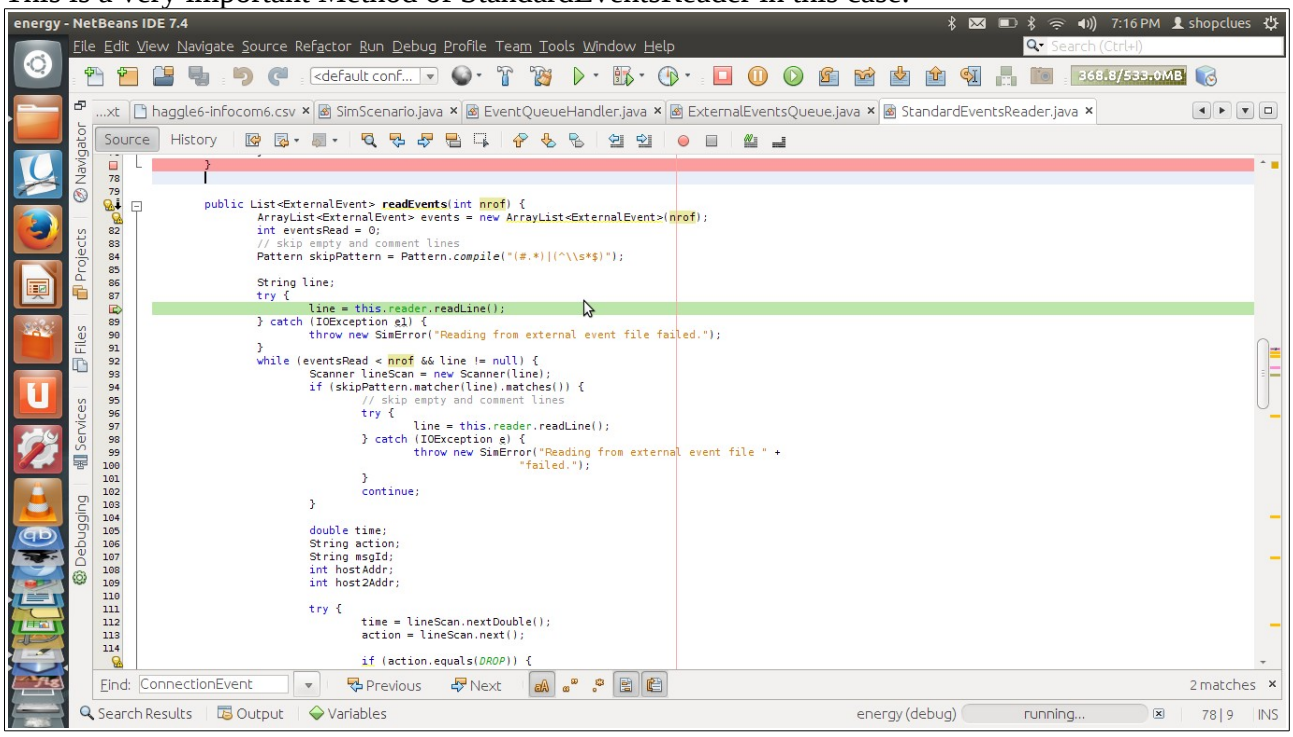
Here it reaches the *this.reader = new StandardEventsReader(eventsFile)* line.

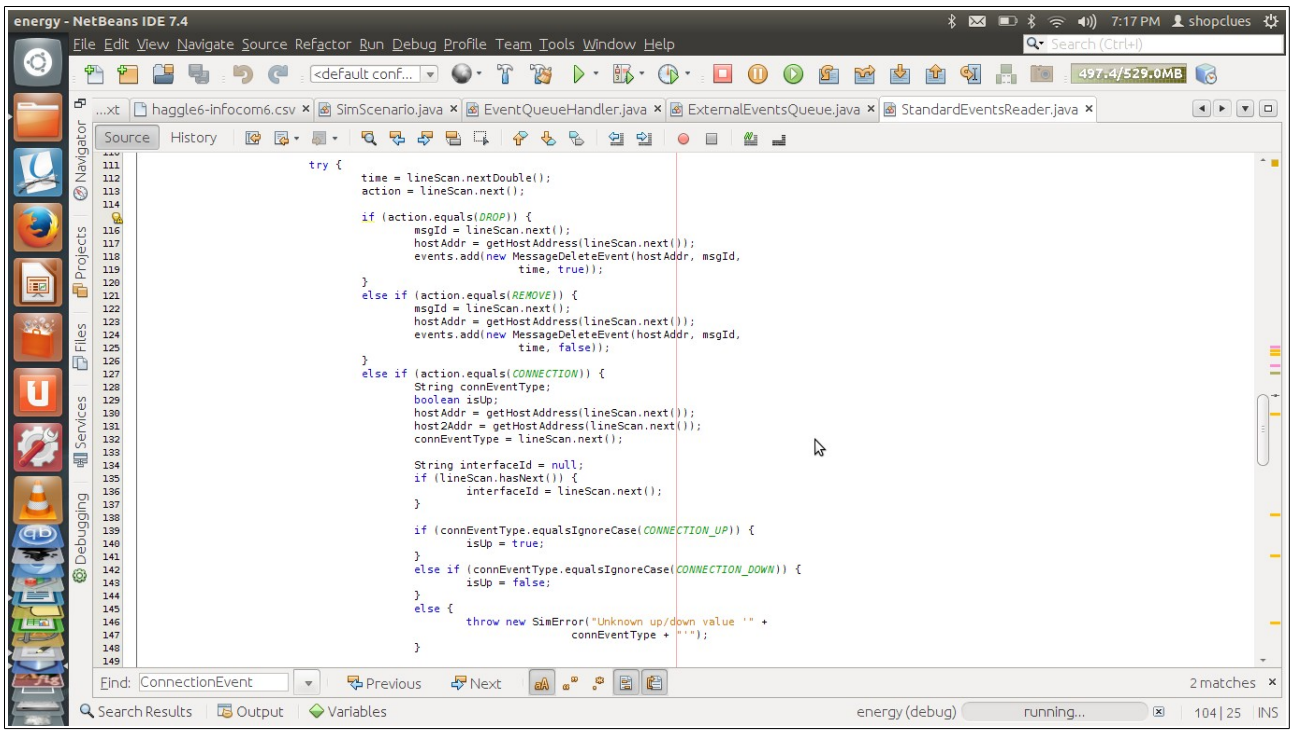
```
        public StandardEventsReader(File eventsFile){
                try {
                        //this.scanner = new Scanner(eventsFile);
                        this.reader = new BufferedReader(new FileReader(eventsFile));
                } catch (FileNotFoundException e) {
                        throw new SimError(e.getMessage(),e);
                }
        }
```

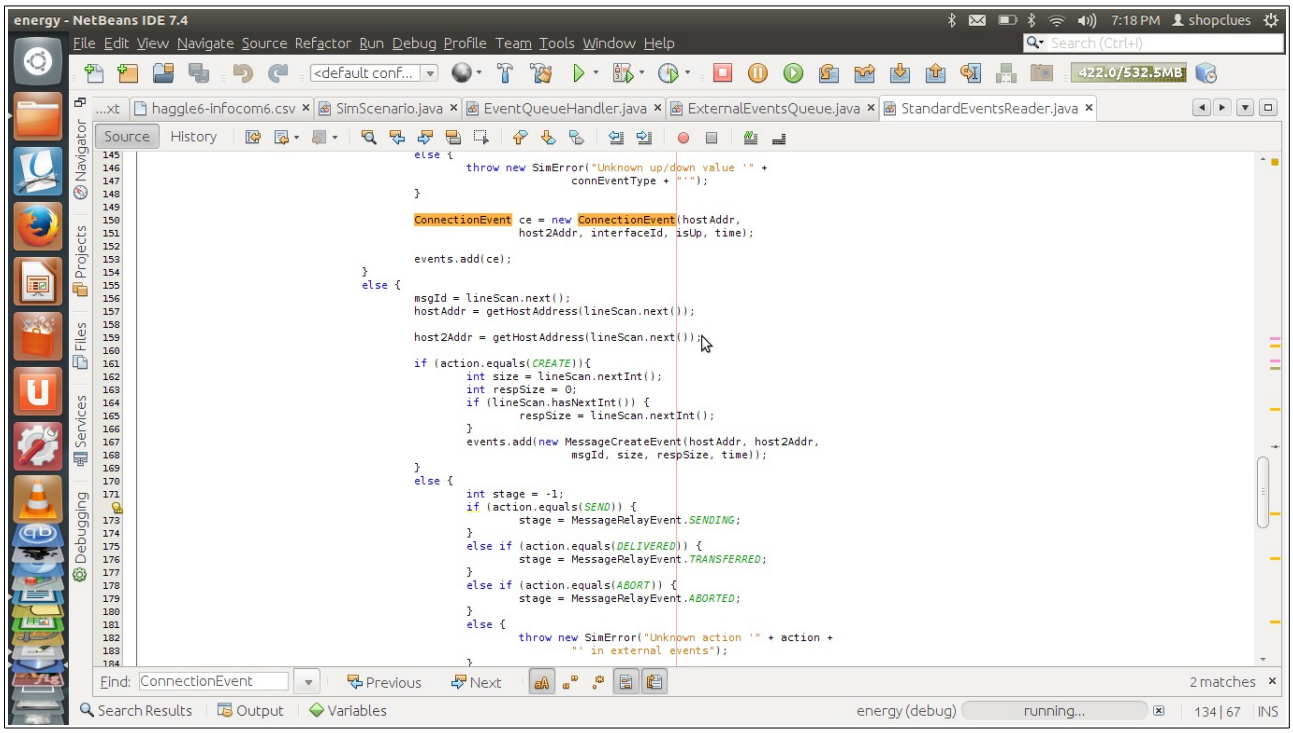
Then *this.queue = readEvents(nrofPreload)* will get executed.



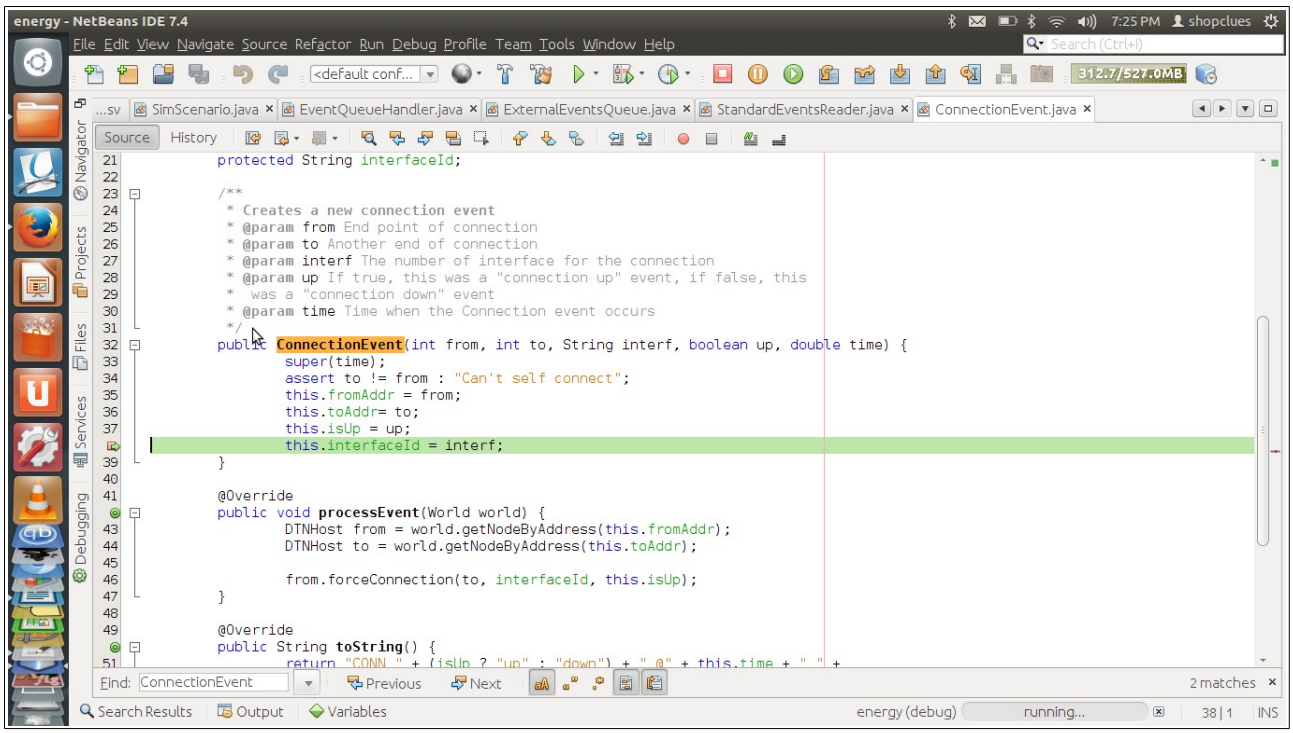
This is a very important Method of StandardEventsReader in this case.





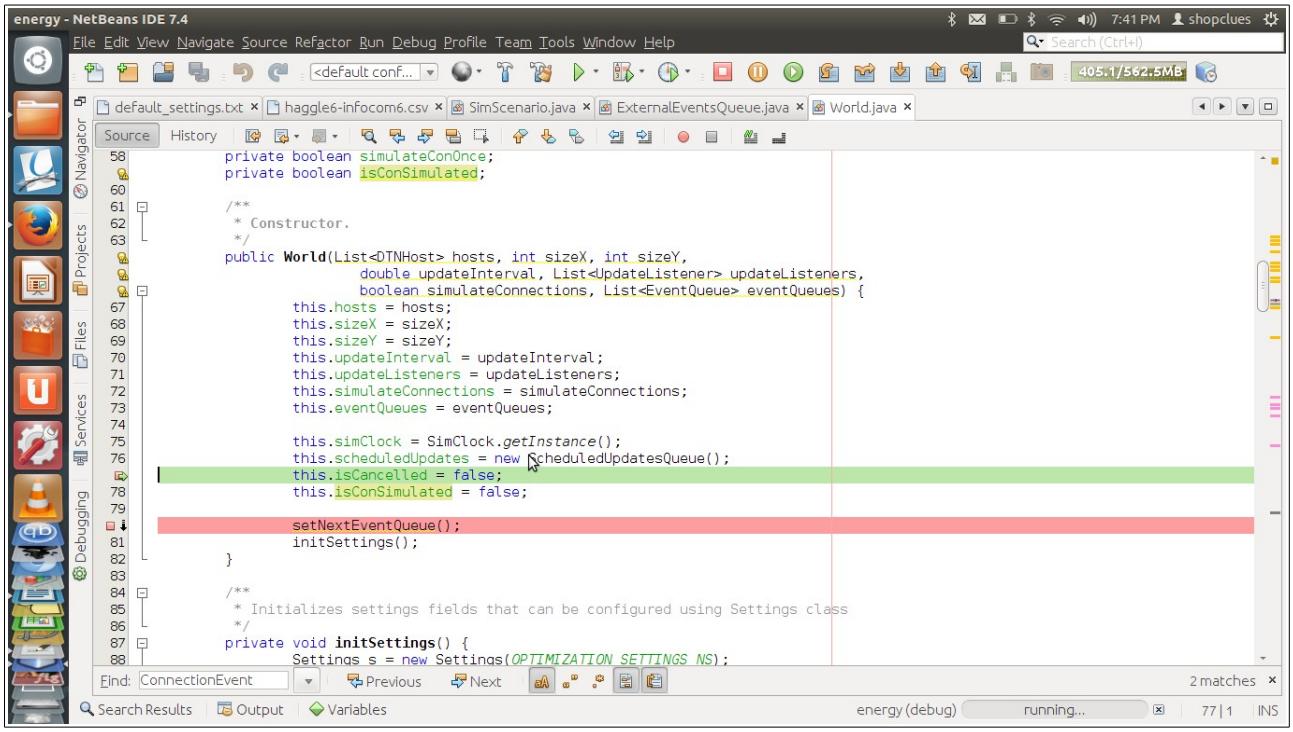


Here heart of External file reading lies. In our Case it reaches to *elseif(action.equals(CONNECTION)) {* and *ConnectionEvent ce = new ConnectionEvent(hostAddr,host2Addr, interfaceId, isUp, time)* is made.



This will return and create events of upto 500 lines of file. Again back to readEvents Method of ExternalEventsQueue Class. Then back to *EventQueueHandler()* constructure. There *queues.add(new ExternalEventsQueue(path, preload))* line execution gets finished. Then it comes back to Simscenario.java constructure where *this.eqHandler = new EventQueueHandler()* line execution gets finished. Now *List<ExternalEvent> events* attribute of ExternalEventsQueue object contains ConnectionEvent objects list.

Lets us see what happens next.
Then it reaches the world constructure.



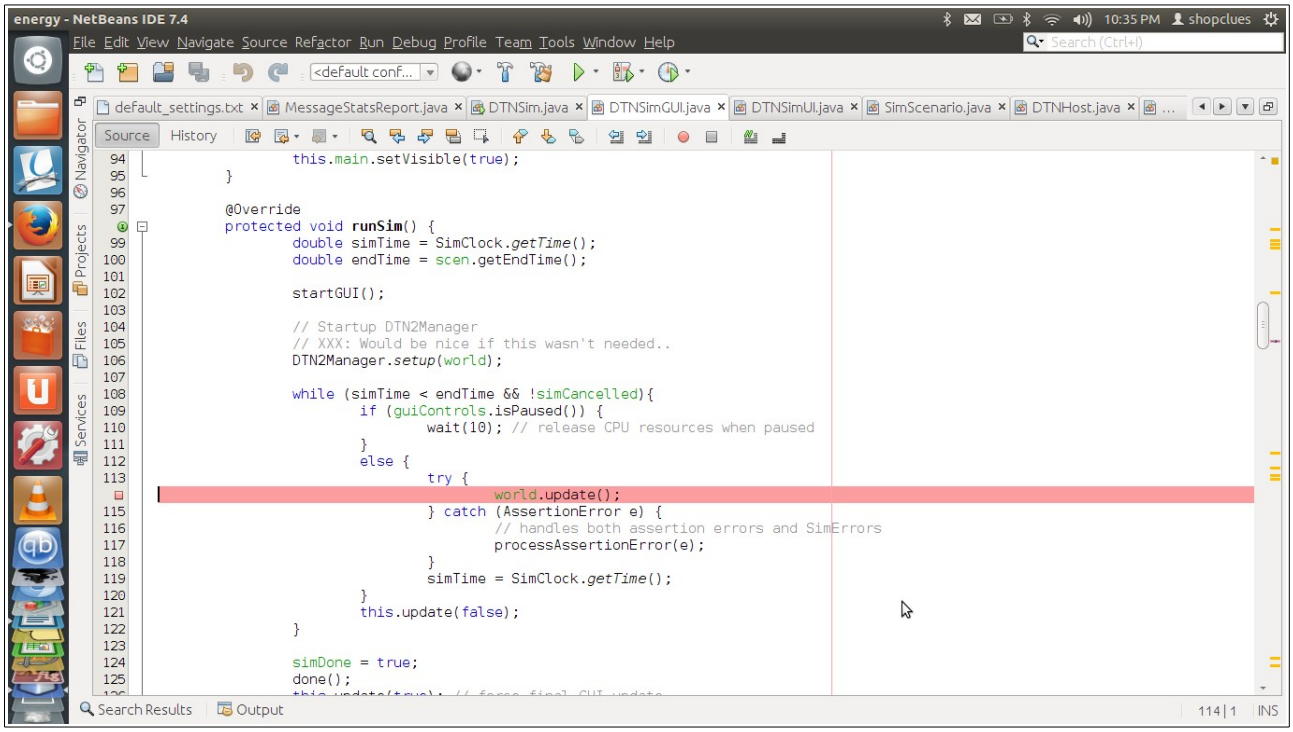
Here eventQueues has ExternalEventsQueue which has queues of event list of ConnectionEvent. Then it comes to *setNextEventQueue().*

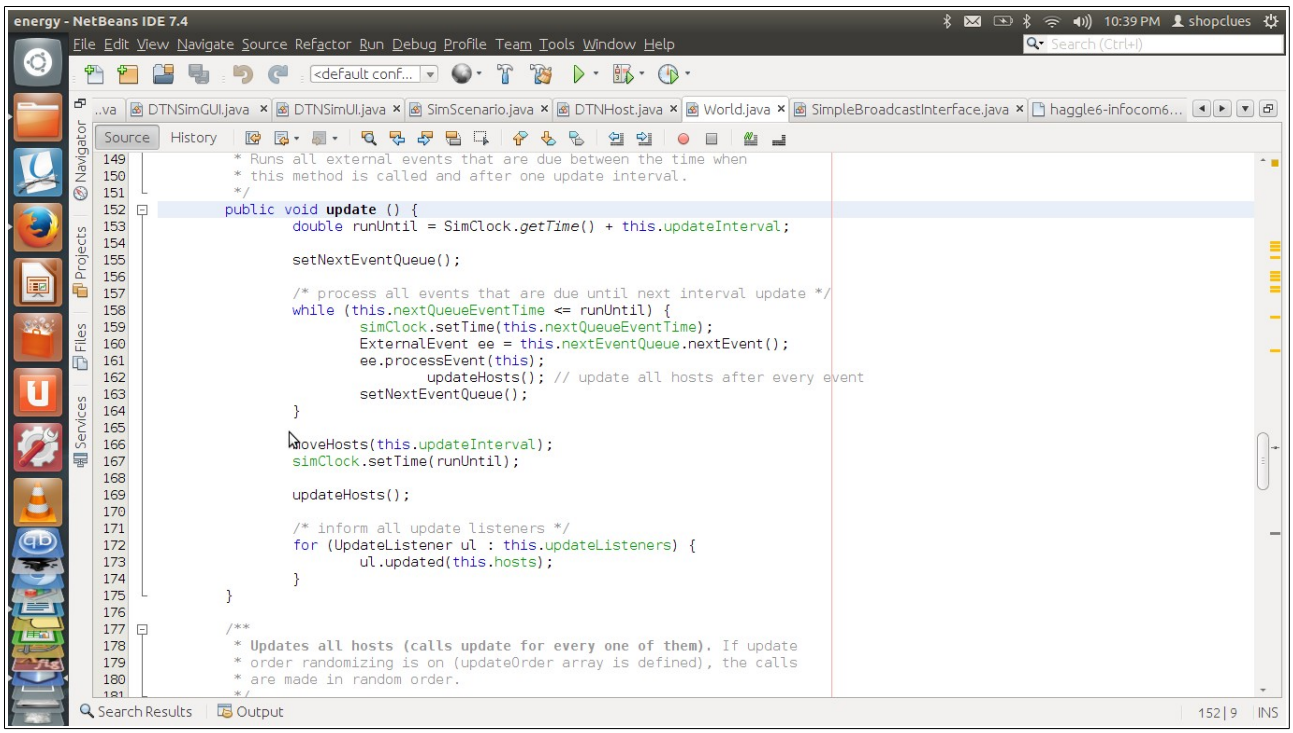
```
        /**
         * Goes through all event Queues and sets the
         * event queue that has the next event.
         */
        public void setNextEventQueue() {
                EventQueue nextQueue = scheduledUpdates;
                double earliest = nextQueue.nextEventsTime();

                /* find the queue that has the next event */
                for (EventQueue eq : eventQueues) {
                        if (eq.nextEventsTime() < earliest){
                                nextQueue = eq;
                                earliest = eq.nextEventsTime();
                        }
                }

                this.nextEventQueue = nextQueue;
                this.nextQueueEventTime = earliest;
        }
```

Here earliest value is 4500 seconds as defined in first line of connection file. this.nextEventQueue has ExternalEventsQueue object in it.

Running Phase working

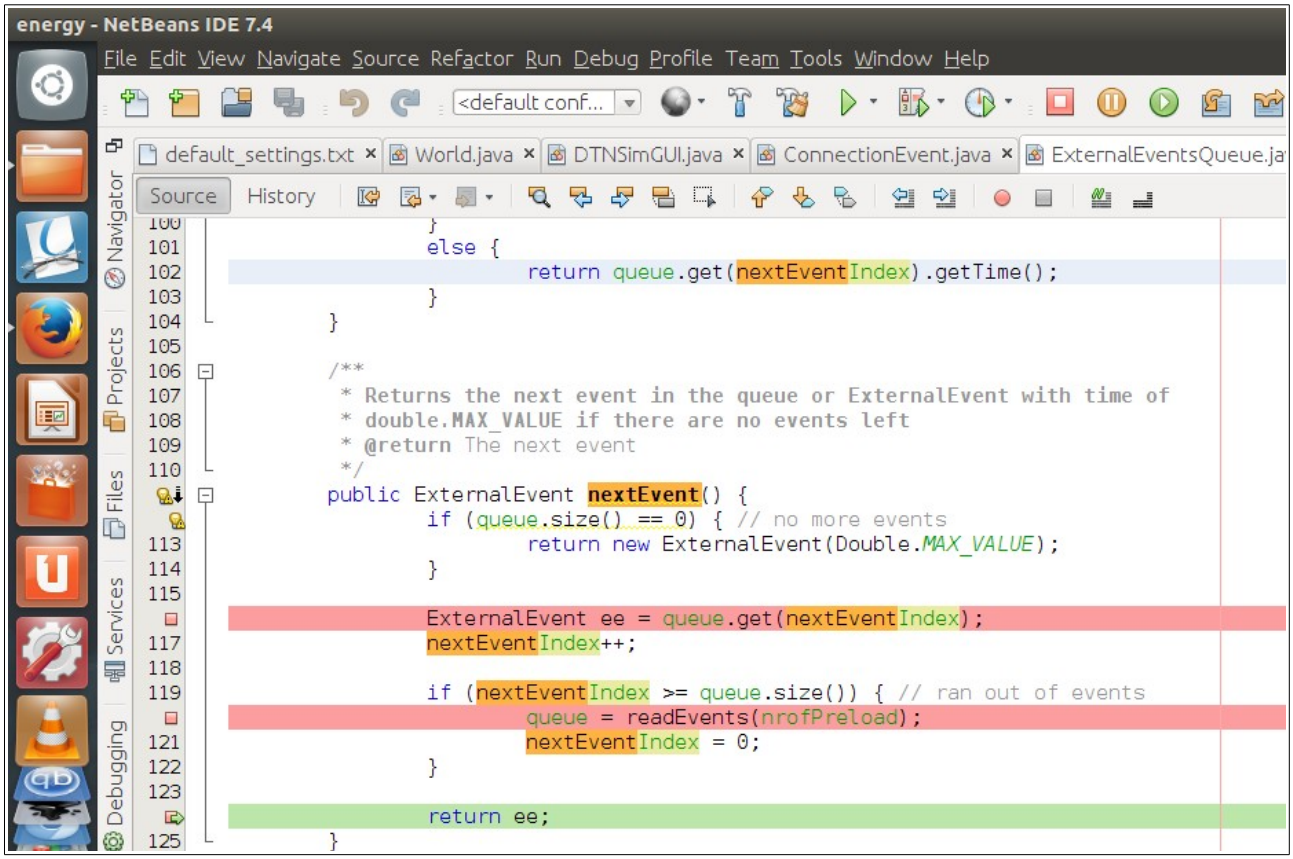
Directly moving to world.update method of in DTNSimGUI runSim Method.



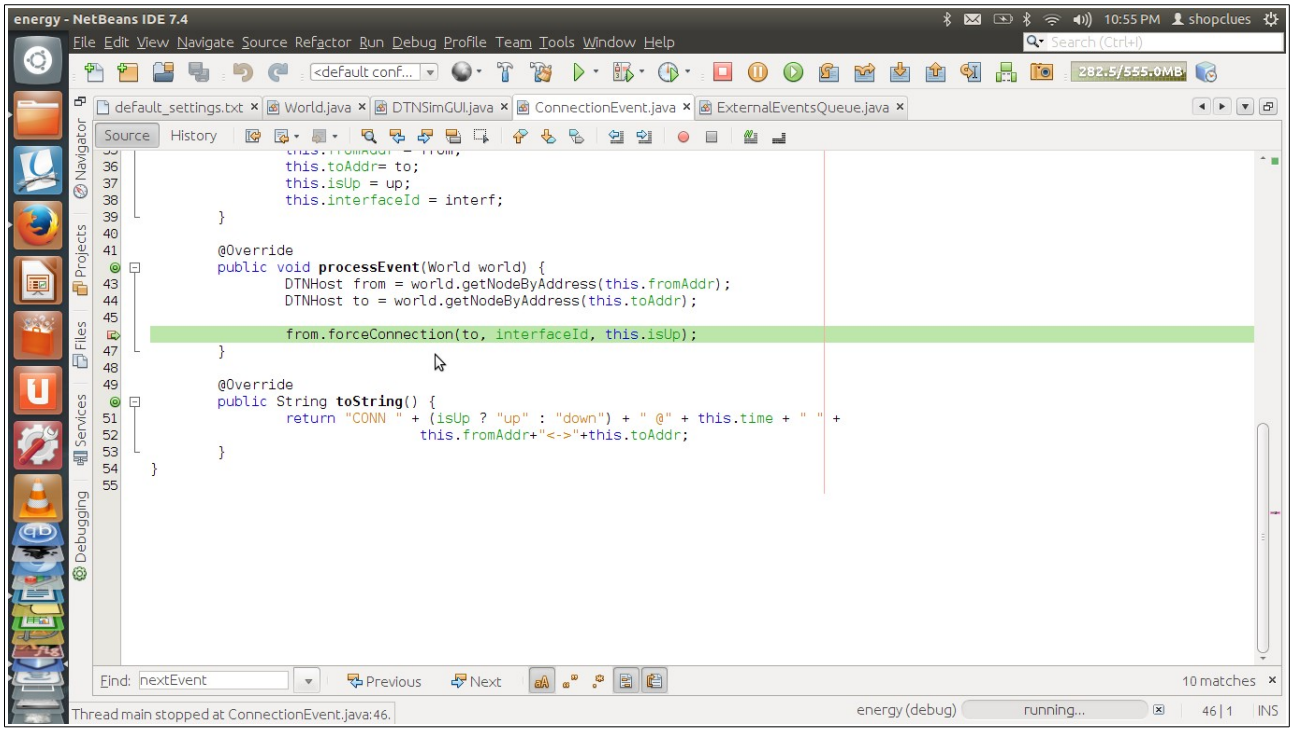


We already see the activity in world update class for the case of the First case study.

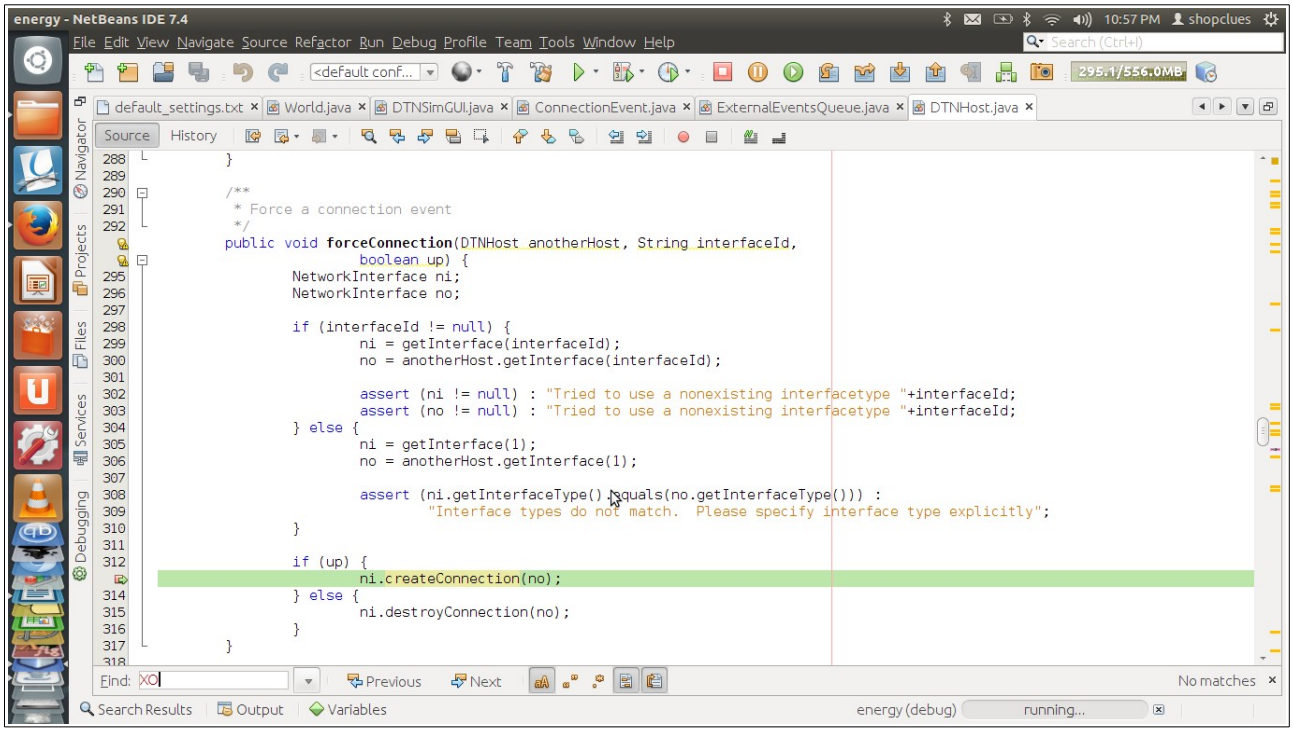
Let us again see for this case study. *setNextEventQueue()* will do same thing as in initialisation phase. But when *while (this.nextQueueEventTime <= runUntil)* will become true then inner code will excecute. *this.nextQueueEventTime* will remain 4500 till *runUntil* will reach 4500.



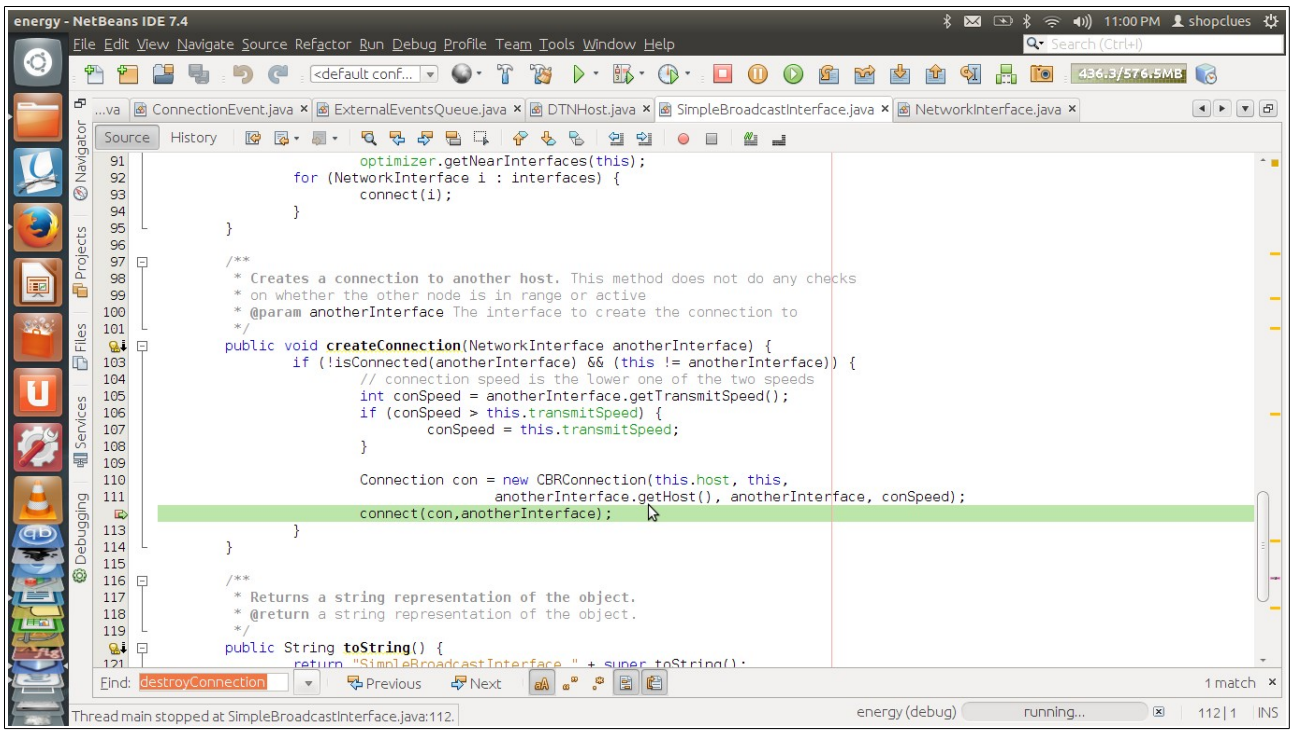
Here ee is ConnectionEvent object that is being returned.



Here forced connections are made or Broken down as per file instruction.

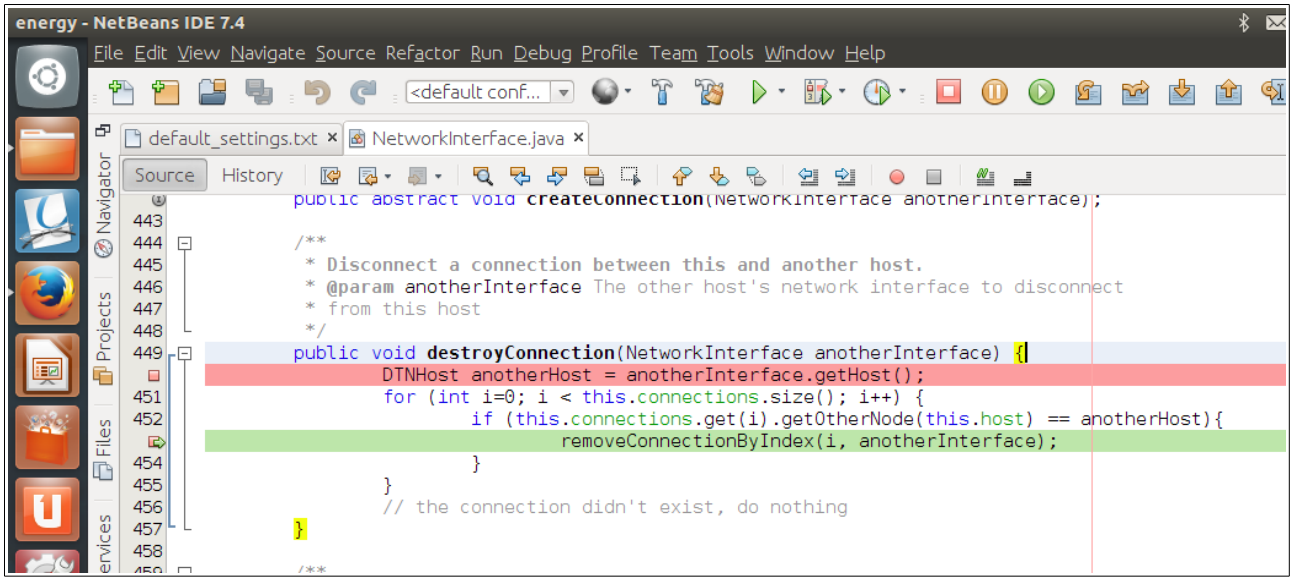


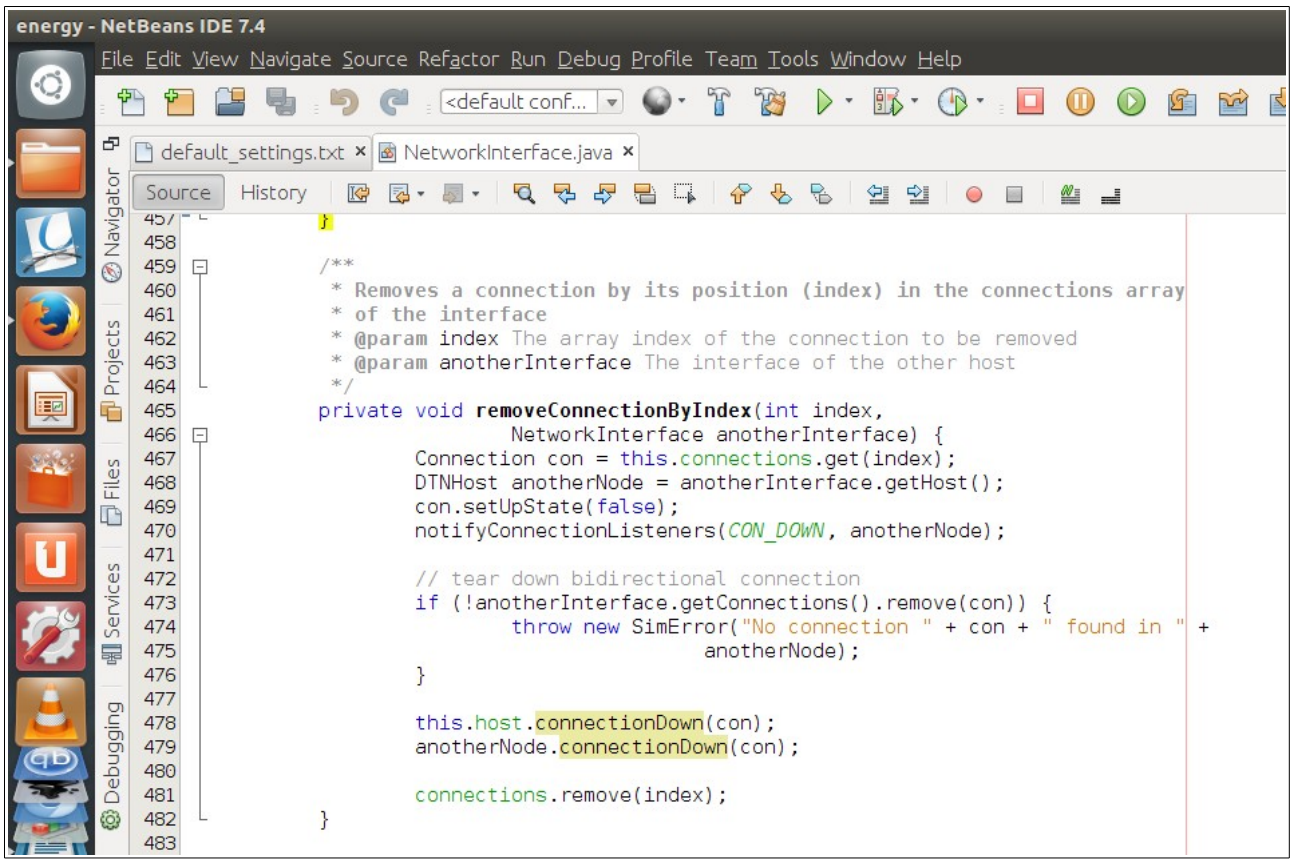
It calls SimpleBroadCastInterface createConnection Method.

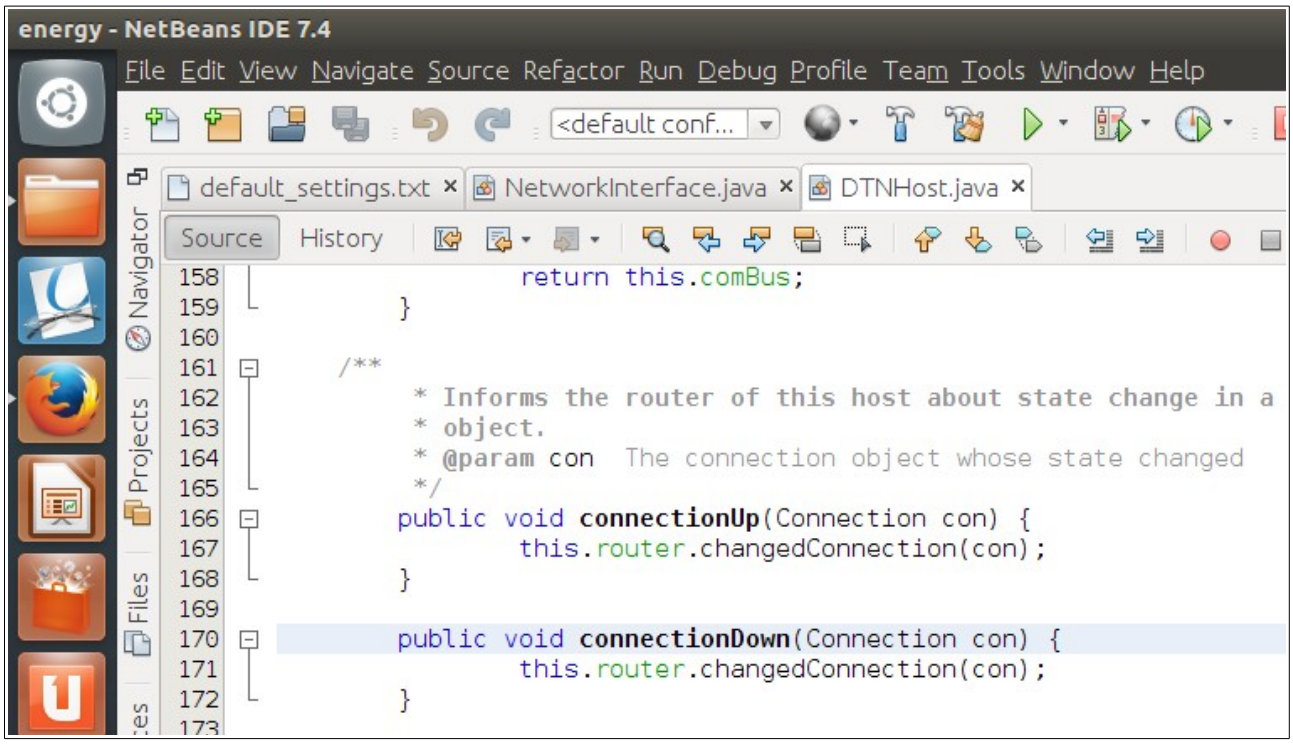


It calls the connect. Rest process is already discussed.

It also calls *ni.destroyConnection(no)* which destroys connections as can be seen in snapshots.







ConnectionUp and ConnectionDown can be used by router to get information of connections.
In this setting Only One Event is used to simulate Connections. If Messages Generation Phenomean have to be added then MessageEventGent Generator can be used or file containing MessageEvent can be used. Sample shown in Appendix section.

Appendix

Default_settings.txt file for CASE I Analysis

```
## Scenario settings
Scenario.name = default_scenario
Scenario.simulateConnections = true
Scenario.updateInterval = 0.1
# 43200s == 12h
Scenario.endTime = 43200


## Interface-specific settings:
# type : which interface class the interface belongs to
# For different types, the sub-parameters are interface-specific
# For SimpleBroadcastInterface, the parameters are:
# transmitSpeed : transmit speed of the interface (bytes per second)
# transmitRange : range of the interface (meters)

# "Bluetooth" interface for all nodes
btInterface.type = SimpleBroadcastInterface
# Transmit speed of 2 Mbps = 250kBps
btInterface.transmitSpeed = 250k
btInterface.transmitRange = 10

# High speed, long range, interface for group 4
highspeedInterface.type = SimpleBroadcastInterface
highspeedInterface.transmitSpeed = 10M
highspeedInterface.transmitRange = 1000

# Define 6 different node groups
Scenario.nrofHostGroups = 6

## Group-specific settings:
# groupID : Group's identifier. Used as the prefix of host names
# nrofHosts: number of hosts in the group
# movementModel: movement model of the hosts (valid class name from movement package)
# waitTime: minimum and maximum wait times (seconds) after reaching destination
# speed: minimum and maximum speeds (m/s) when moving on a path
# bufferSize: size of the message buffer (bytes)
# router: router used to route messages (valid class name from routing package)
# activeTimes: Time intervals when the nodes in the group are active (start1, end1, start2, end2, ...)
# msgTtl : TTL (minutes) of the messages created by this host group, default=infinite

## Group and movement model specific settings
# pois: Points Of Interest indexes and probabilities (poiIndex1, poiProb1, poiIndex2, poiProb2, ... )
#       for ShortestPathMapBasedMovement
```

```
# okMaps : which map nodes are OK for the group (map file indexes), default=all
# for all MapBasedMovent models
# routeFile: route's file path - for RandomWaypoint
# routeType: route's type - for RandomWaypoint


# Common settings for all groups
Group.movementModel = RandomWaypoint
Group.router = ProphetRouter
Group.bufferSize = 10M
Group.waitTime = 0, 120
# All nodes have the bluetooth interface
Group.nrofInterfaces = 1
Group.interface1 = btInterface
# Walking speeds
Group.speed = 0.5, 1.5
# Message TTL of 300 minutes (5 hours)
Group.msgTtl = 300
Group.nrofHosts = 58

####
Group.initialEnergy = 4800
Group.scanEnergy = 0.06
Group.scanResponseEnergy = 0.08
Group.transmitEnergy = 0.08
Group.baseEnergy = 0.07
Group.charging_cofficient = 20
Group.thrushold_energy = 3000
####

# group1 (pedestrians) specific settings
Group1.groupID = p

# group2 specific settings
Group2.groupID = c
# cars can drive only on roads
Group2.okMaps = 1
# 10-50 km/h
Group2.speed = 2.7, 13.9

# another group of pedestrians
Group3.groupID = w

# The Tram groups
Group4.groupID = t
Group4.bufferSize = 50M
Group4.movementModel = RandomWaypoint
Group4.routeFile = data/tram3.wkt
Group4.routeType = 1
Group4.waitTime = 10, 30
```

```
Group4.speed = 7, 10
Group4.nrofHosts = 2
Group4.nrofInterfaces = 2
Group4.interface1 = btInterface
Group4.interface2 = highspeedInterface

Group5.groupID = t
Group5.bufferSize = 50M
Group5.movementModel = RandomWaypoint
Group5.routeFile = data/tram4.wkt
Group5.routeType = 2
Group5.waitTime = 10, 30
Group5.speed = 7, 10
Group5.nrofHosts = 2

Group6.groupID = t
Group6.bufferSize = 50M
Group6.movementModel = RandomWaypoint
Group6.routeFile = data/tram10.wkt
Group6.routeType = 2
Group6.waitTime = 10, 30
Group6.speed = 7, 10
Group6.nrofHosts = 2


## Message creation parameters
# How many event generators
Events.nrof = 1
# Class of the first event generator
Events1.class = MessageEventGenerator
# (following settings are specific for the MessageEventGenerator class)
# Creation interval in seconds (one new message every 25 to 35 seconds)
Events1.interval = 25,35
# Message sizes (500kB - 1MB)
Events1.size = 500k,1M
# range of message source/destination addresses
Events1.hosts = 0,180
# Message ID prefix
Events1.prefix = M

## Movement model settings
# seed for movement models' pseudo random number generator (default = 0)
MovementModel.rngSeed = 1
# World's size for Movement Models without implicit size (width, height; meters)
MovementModel.worldSize = 4500, 3400
# How long time to move hosts in the world before real simulation
MovementModel.warmup = 1000

## Map based movement -movement model specific settings
MapBasedMovement.nrofMapFiles = 4
```

```
MapBasedMovement.mapFile1 = data/roads.wkt
MapBasedMovement.mapFile2 = data/main_roads.wkt
MapBasedMovement.mapFile3 = data/pedestrian_paths.wkt
MapBasedMovement.mapFile4 = data/shops.wkt

## Reports - all report names have to be valid report classes

# how many reports to load
Report.nrofReports = 2
# length of the warm up period (simulated seconds)
Report.warmup = 0
Report.granularity = 43000
# default directory of reports (can be overridden per Report with output setting)
Report.reportDir = reports/
# Report classes to load
Report.report1 = MessageStatsReport
Report.report2 = EnergyLevelReport


## Default settings for some routers settings
ProphetRouter.secondsInTimeUnit = 30
ProphetRouter.alpha = 0.5
ProphetRouter.beta = 0.5
ProphetRouter.energy_max = 4800

SprayAndWaitRouter.nrofCopies = 6
SprayAndWaitRouter.binaryMode = true

## Optimization settings -- these affect the speed of the simulation
## see World class for details.
Optimization.cellSizeMult = 5
Optimization.randomizeUpdateOrder = true


## GUI settings

# GUI underlay image settings
GUI.UnderlayImage.fileName = data/helsinki_underlay.png
# Image offset in pixels (x, y)
GUI.UnderlayImage.offset = 64, 20
# Scaling factor for the image
GUI.UnderlayImage.scale = 4.75
# Image rotation (radians)
GUI.UnderlayImage.rotate = -0.015

# how many events to show in the log panel (default = 30)
GUI.EventLogPanel.nrofEvents = 100
# Regular Expression log filter (see Pattern-class from the Java API for RE-matching details)
#GUI.EventLogPanel.REfilter = .*p[1-9]<->p[1-9]$
```

Extra default file setting for CASE II analysis

Events.nrof = 2
Events1.class = StandardEventsReader
Events1.filePath = <<directory containing the trace file>>/mytracefile.txt

Events2.class = MessageEventGenerator
# change interval to have more or less messages, see javadocs for more information.
Events2.interval = 30,40
Events2.size = 1k
# range of message source/destination addresses
Events2.hosts = <<min node ID>>,<<max node ID>>
Events2.prefix = M

In this case, *setNextEventQueue()* method of world will use MessageEventGenerator also as in first case study to set Message in the nodes. This is also set from *EventQueueHandler()* constructure call from Simscenario intialisation.

Discussion on StandardEventsReaders.java
This class is specifically designed for events readings. Reading this file and special focus on *public List<ExternalEvent> readEvents(int nrof)* method will explain all.

Sample file of CASE II Analysis

Portion of ***haggle6-infocom6.csv*** used in our test case.

4500.00 CONN 16 12 up
5623.00 CONN 12 16 up
5872.00 CONN 2 13 up
5960.00 CONN 15 52 up
6010.00 CONN 13 2 up
6023.00 CONN 4 3 up
6207.00 CONN 21 24 up
6639.00 CONN 55 24 up
6802.00 CONN 84 90 up
6848.00 CONN 3 4 up
6850.00 CONN 4 0 up
6900.00 CONN 0 39 up
6915.00 CONN 84 95 up
6942.00 CONN 86 50 up
7025.00 CONN 15 16 up
7032.00 CONN 45 82 up
7033.00 CONN 84 67 up
7035.00 CONN 66 94 up
7035.00 CONN 69 40 up
7045.00 CONN 21 55 up
7128.00 CONN 0 51 up

7146.00 CONN 85 70 up
7152.00 CONN 92 63 up
7156.00 CONN 89 92 up
7159.00 CONN 64 94 up
7163.00 CONN 44 37 up
7163.00 CONN 44 77 up
7170.00 CONN 27 95 up
7173.00 CONN 86 41 up
7205.00 CONN 4 15 up
7217.00 CONN 70 86 up
7255.00 CONN 14 37 up
7259.00 CONN 85 20 up
7259.00 CONN 88 22 up
7259.00 CONN 88 41 up
7261.00 CONN 12 1 up
7262.00 CONN 15 12 up
7267.00 CONN 71 80 up
7268.00 CONN 68 53 up
7269.00 CONN 68 39 up
7269.00 CONN 94 53 up
7269.00 CONN 95 36 up
7270.00 CONN 61 35 up

Sample section of file containing MessageEvent List

3662.5	C	M1	4	2	1262799
3721.7	C	M2	90	76	1292645
3730.6	C	M3	33	99	1236983
3805.4	C	M4	62	12	1191810
3817.3	C	M5	67	19	606629
3824.3	C	M6	1	73	1221142
3829.0	C	M7	19	13	291138
3833.8	C	M8	59	71	1633127
3836.6	C	M9	38	26	1333674
3843.8	C	M10	45	99	446071
3911.7	C	M11	16	32	1302999
3931.6	C	M12	98	55	355706
937.4	C	M13	71	47	1461414
3971.9	C	M14	89	85	1740698
3979.8	C	M15	10	53	368358
3982.3	C	M16	4	65	1403045