SYSC 5104

Discrete Event Modelling and Simulation

Assignment2: Modelling and Simulation of an Ad-Hoc Network

By: Monageng Kgwadi

SN: 100626879

Date: 29 October 2007

Ad-hoc network modelling and simulation

In this assignment, the system to be modelled is a wireless ad-hoc network. The goal of the exercise is to model and simulate an ad-hoc network in which nodes can arbitrarily enter or leave the system. The nodes move about and the connections between them keep changing, thus the topology of the network keeps on changing. After the system is modelled, a simulation on the reliability of the network will be run to determine the reliability of such a network to sustain communications between hosts with capacity constraints on the links (i.e. the links must not only exists but capable of supporting the capacity the communication requires).
The network will be modelled in DEVS as a coupled model which consists of components that model nodes and links, the diagram below shows a preliminary sketch of the system

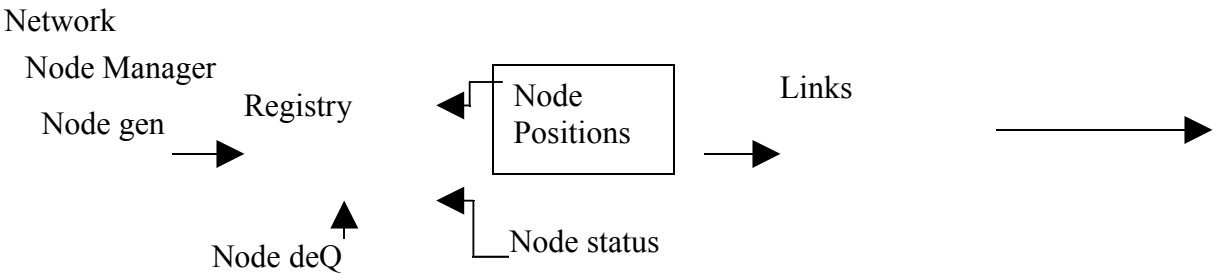

Figure1: Conceptual Model

The *Node gen* component is an atomic model that generates nodes at random to put into the network, the generated nodes are assigned arbitrary positions on the network. State variables: state ={ generate, idle} /*calculates the time to add a node to network */
The *Node deQ* component arbitrarily removes a node from the network with a predefined probability. State Variables: state = {remove, idle}
The *Registry* keeps the nodes that are in the network and removes some nodes at random. State variables: Phase = {idle, calc node positions, calc node status,..} capacity /* maximum number of nodes */; Nodes = none<nodeID, position, status> /* the nodes in network with ID, location and status*/
The *Node Position* component periodically computes the positions of all nodes in the Registry based on the likelihood of each node to move given by a known probability. State variables: phase = {calculate, idle} /* calculate the new positions of nodes*/
The *Node Status* component periodically assigns the working status of a link based on a predetermined probability that a node will work or fail. State variables: phase ={calculate, idle} /* calculate the new status of nodes after some time*/

The *Node manager* manages the nodes and takes care of entry and exit of nodes into the network, node failures and node movements.

The *Links* component determines the existence of a working link between nodes as a function of separation distance and demanded link capacity. State variables: Links=/* matrix of all working links between nodes*/

The Link component will be modelled and incorporated into the system if and when time permits to keep the complexity of the modelling manageable.

Defining the model using DEVS specifications

The objective of this part of this assignment is to use DEVS formalism to define a model of a simple adhoc network. An adhoc network is a network in which nodes are connected by wireless links and can freely move about. Moreover, at any time there could be nodes entering or leaving the network. Thus the topology of the network is constantly changing. The links between the nodes is a strong function of the separation distance between the nodes. A communication link between any two nodes exist if there is a direct link between the two nodes or there is an indirect link through other node(s).

Conceptual Model

In this conceptial model of an adhoc network, the nodes are confined to an area of size xmax by ymax meters. The position of a node in the network is given by its cartesian coodinates(x,y) in the area. A node also has a speed s, associated with it which determines how fast the node travels to its new destination position, (Xd,Yd). Moreover, a node has a variable that specifies the status of the node, i.e. the node is working or has failed.
The network model consists of a registry which keeps a collection of the nodes in the network. Also in the registry model, is an array of N by N which keeps the status of links between nodes; where N is the maximum number of nodes allowed in the network at a time. A link exists between two nodes if the separation distance between the nodes is at most the transmission range, Tr. The model assumes links are symmetrical i.e. the link from node A to B is the same as from node B to A. Thus the array of links needs only one entry for the two nodes A and B to characterise if the link exists between the two nodes. The registry periodically updates the positions of the nodes and the link status, this mimics the motion of the nodes and hence the changing topology of the network. To mimic the random entry and exit of nodes into the network, the registry randomly adds and removes nodes from the collection of nodes and updates the array of links.

DEVS formalism definition
The network is defined using the DEVS formalism as follows based on the picture below.

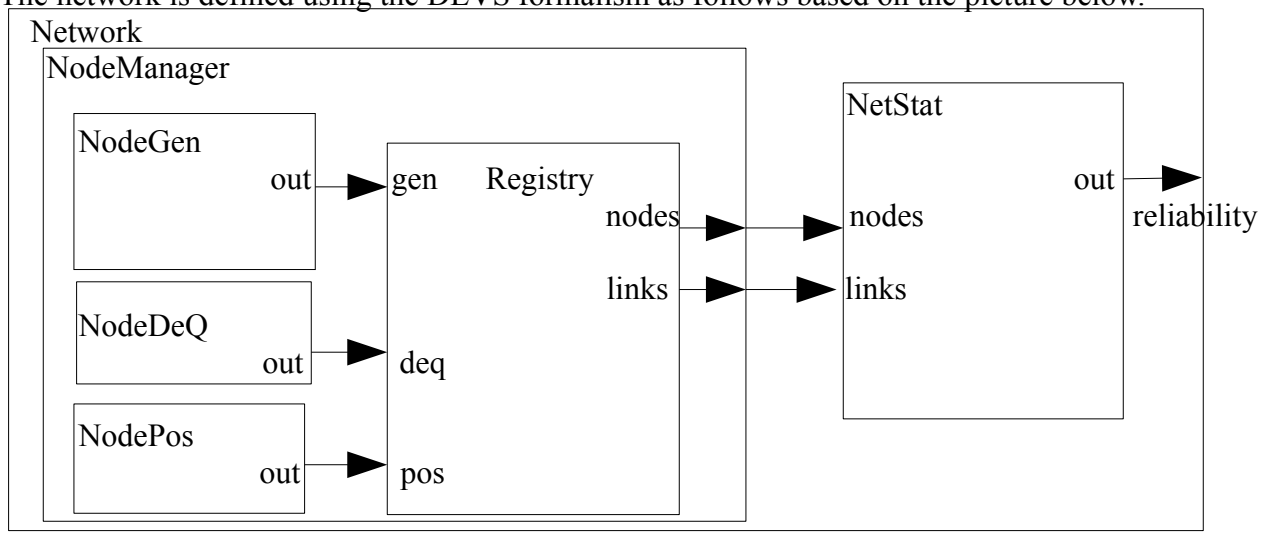

Figure 2: Adhoc Network model
The model Network is a coupled model and defined as:

Network = <X,Y,D,M, EIC,EOC,IC,select>

X = {Ø};

Y = {(reliability, **R**□ [0,1])};

D = {NodeManager, NetStats};

Md = {Mnodemanager, Mnetstats};

EIC = {Ø};

EOC = {(NetStats.out , reliability)};

IC = {(NodeManager.nodes, NetsStats.nodes) ,(NodeManager.links, NetsStats.links) };

select = {NodeManager, NetStats};

The NetStatistics is an atomic model that calculates the perfomance of the network and can be defined as:

NetStatistics = <X, Y ,S, δext, δint ,λ, ta >

X= {(nodes, **N**), (links,**N**)};

Y = {(out,**R**□ [0;1])};

S = {state □{active,passive}, reliability, **R**□ [0,1]};

```
δext(S, e, x){
        if(x.port == nodes){
                update the reliability using the updated number of nodes in the network;
                state = active;
                ta(state, Zero);
                }
        if(x.port == links){
                update the reliability of the network using the updated links information;
                state = active;
                ta(state, Zero);
                }
}
δint(InternalMessage){
        passivate();
}
λ(InternalMessage &Msg){
sendOutput(time,out,reliability);  //ends the relaibility of the network in realtime.
}
```

The NodeManager is a coupled model defined as:

NodeManager =<X,Y,D,M, EIC,EOC,IC,select>

X = {Ø};

Y = {(links,**N**) , (nodes,**N**)};

D = {NodeGen, NodeDeQ, NodePositions, Registry};

Md = {MnodeGen , Mnodepositions, Mnodedeq, Mregistry};

EIC = {Ø};

EOC = {(Registry.nodes, Self.nodes), (Registry.links, Self.links)};

IC = {(NodeGen.out, Registry.gen), (NodeDeQ.out, Registry.deq),
(NodePositions.out, registry.pos)};

select = {Registry, NodePositions, NodeDeQ, NodeGen};

The Registry model is an atomic model defined as:

Registry =<X, Y ,S, δext, δint ,λ, ta >

X = {(gen,binary) , (deq,binary),(pos, binary)};

Y = {(nodes, N), (links,N)};

S = { state □{active,passive}; xmax, ymax, nodes □ **N**;maxSpeed □ **R**+;
nodes {px,py, dx,dy□ **N**; status □ bool; speed □ **R**[0, maxSpeed] }*;
links{linkStatus □ bool }*}

δext (S,e,x){

if(x.port == gen && x.value ==1) {
create a new node with random coordinates, status, speed and add to node array;
update the links array;
ta(0);
}

if(x.port == deq && x.value ==1) {
remove a node at random from the array;
update the links array;
ta(0);
}

if(x.port == pos && x.value ==1) {
update positions of nodes using: new pos = speed*timeElapsed;

if (destination reached) randomly select new speed and destination;
assign a random node status; // albeit keeping a predifine % availability (eg 5% failure)
update the links array;
ta(0);
}
}

δint(InternalMessage &msg){
passivate;
}
λ(InternalMessage &msg){
sendOutput(msg.time(),links, |links|);
sendOutput(msg.time(),nodes, |nodes|);
}

The NodeGen and NodeDeQ models are instances of a Generator atomic model defined as:

Generator = <X, Y ,S, δext, δint ,λ, ta >

X = {Ø};

Y = {(out,binary)};

```
S = {state □ {active, passive}, distribution □ Distribution};
δext(S, e, x){  };
δint(InternalMessage &msg){
        holdIn(active, Time(  fabs(distribution().get())  )  ); //generate after a some time
                                                                //described by the distrubution
}
λ(InternalMessage &msg){
        sendOutput(msg.time(), out, 1);
}
```

The NodePositions model produces an output after a predetermined and fixed time to instruct the node registry model to update the positions of nodes and the status of the links. It is similar to the Generator model described above by construction, the difference being the state variables and the internal transition function. The model description is given below.

```
NodePositions = <X, Y ,S, δext, δint ,λ, ta >
        X = {Ø};
        Y = {(out,binary)};
        S = {state □ {active, passive}, const Time updateTime};
        δext(S, e, x){  };
        δint(InternalMessage &msg){
                holdIn(active, updateTime  );
        }
        λ(InternalMessage &msg){
                sendOutput(msg.time(), out, 1);
        }
```

Proposed testing of models and integration

The proposed testing method involves building the simpliest atomic model, the Generator model and the testing it by creating a top model and testing it. Since the generator model is used to randomise entry and exit of network nodes, the test will be to observe if the outputs of the model are random. The random behavour expected in this case is a linear time delay between outputs, given an upperbound on the delay.The NodePositions model will be tested in the same manner but in this case the output observed should have equal fixed delay between them.
The Registry model will be developed in stages and tested at each stage. To start off, the model will be built only to get network nodes to enter and leave the registry. To test this an event file will be created to request the entry and exit of nodes and the output will be observed to confirm the number of nodes in the network after each change. After the initial test has passed, the registry will be implemented to update the positions of network nodes as time advances according to the speed and direction. This part will be tested by having the details of nodes output on the console(or file) each time such updates are made.

The last part to be implemented will be the link status in the registry. This part will tested by having the link status matrix output on a console and backchecking each link status with the information of each node on the network to confirm the validity of each link. For a link to exist between any two nodes, the separtion between the two nodes has to be within a predefined distance of coverage. The final test would be to combine all the above mentioned tests and validate that they all hold.

After all the components that make up the NodeManager are individually tested, they will be coupled together to define the NodeManager model and tested. The test at this point will be to observe the overall functinality of the network. The behavior seen at this point should be ad-hoc, with nodes leaving and entering the network randomly, nodes moving around and the links between the nodes changing due to the movement of nodes.
After the ad-hoc behaviour has been modelled satisfactorily, the NetStat model will be built. This model will be constructed to collect the network statistics of interest based on the information of the network. The statistics of interest at this time is the availabity of a communication link between any two randomly picked nodes in the network. The value will be presented as a probability that the link exist. Different network parameters would be varied to observe how they affect the availability of such links. To test the NetStat model, the link matrix would be used as the source of the data and the availabity calculated by hand and compared with the output of the model.

Testing Cases of the model

The NodeGenerator model was implemented as an atomic model that produces an output after random interval. The builtin uniform function of C++ random library was used, scaled by the upper threshold of the maximum desired time. The test case obsereved for this model was to verify that the output of the model was indeed a uniformly distributed random variable between 0 and the maximum time delay set to 5 minutes. The model is used both to add nodes to the network and remove nodes from the network. The results observed from simulating the model verified the correctness of the model and are presnted below:

```
00:00:00:000 out1 1
00:00:04:000 out1 1
00:00:04:000 out1 1
00:00:08:000 out1 1
```

figure 3: Showing the results of the NodeGenerator model

The NodePosition model implemented to generate an output at fixed time interval was also created and tested following the above template. This model was designed however to produce an output to update the positions of nodes every 0.5 seconds. The results are shown in the figure below, and they verify the model works well because every half second an output of 1 is generated at the output port.

```
00:00:00:500 out1 1
00:00:01:000 out1 1
00:00:01:500 out1 1
00:00:02:000 out1 1
00:00:02:500 out1 1
00:00:03:000 out1 1
00:00:03:500 out1 1
00:00:04:000 out1 1
00:00:04:500 out1 1
00:00:05:000 out1 1
```

Figure 4: showing results of simulating the NodePositions output

The registry was designed to contain network nodes and keep the status of the links between them. In the preliminary design, (part i) the links were proposed to be kept by a separate model but communicating the details of links required a lot of messages. As an attempt to keep things simple the

links were also held in the same container that held the nodes because updating links is easier. The registry model keeps nodes as structures that have position, speed, destination and status. The model has input ports nodes, pos, deq. When an input is recieved by the nodes port, a node object is added to the list of nodes already in the network. This part was designed first and tested, the output of the registry which gives the count of network nodes in the was observed to work as presented in the figure below that shows test cases for the registry model. The next functionality was to implement node exit from the noetwork when an input was recieved from the deq port. In this event a network node was removed at random from the network nodes in the network. This model model also updates the positions of nodes as a function of speed, current position and destination position. To test this case the positions of the nodes were output on the screen to give a visual update of where and how nodes move within the network. Note port out outputs the number of nodes while out2 outputs the number of links

```
00:00:00:000 out 8
00:00:00:000 out2 0
00:00:01:000 out 8
00:00:01:000 out2 9
00:00:02:000 out 8
00:00:02:000 out2 9
00:00:03:000 out 8
00:00:03:000 out2 7
00:00:04:001 out 8
00:00:04:001 out2 9
00:00:05:006 out 8
00:00:05:006 out2 9
00:00:06:010 out 8
00:00:06:010 out2 9
```

Figure 5: showing the results of registry model

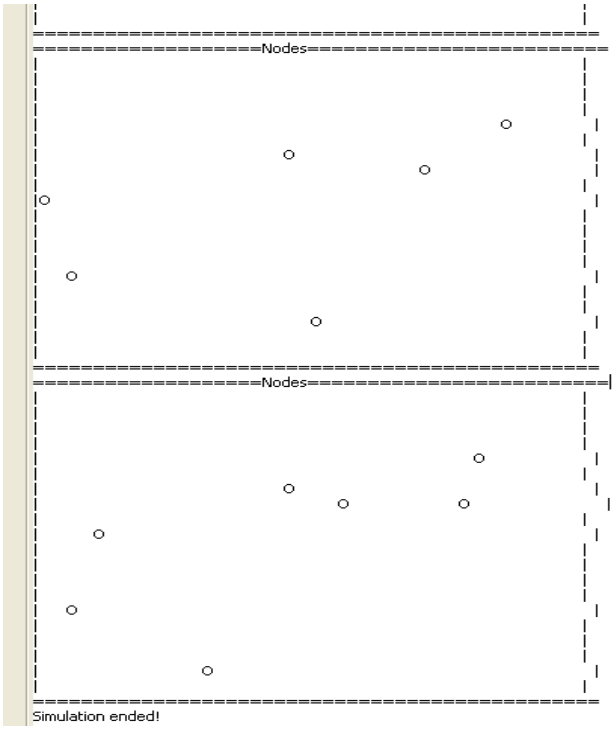

figure 6: showing the positions of nodes on the console (screen captures)

The NodeManager model was created by coupling the two instances of the NodeGenerator model (nodegen and nodeDeQ) , the NodePostions model and the Registry model. The NodeManager does not require an event file since it has no internal inputs and produces outputs to show the number of nodes in the network and the number of links in the network. The verification of the NodeManager is better viewed from the log file to see the interaction between the components with regard to the messages passed between them. The figure below shows part of the log file showing nodes entering and exiting the network.

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
Mensaje I / 00:00:00:000 / top(01) para nodep(02)
Mensaje I / 00:00:00:000 / top(01) para nodegen(03)
Mensaje I / 00:00:00:000 / top(01) para deque(04)
Mensaje I / 00:00:00:000 / top(01) para registry(05)
Mensaje D / 00:00:00:000 / nodep(02) / 00:00:00:500 para top(01)
Mensaje D / 00:00:00:000 / nodegen(03) / 00:00:00:000 para top(01)
Mensaje D / 00:00:00:000 / deque(04) / 00:00:00:000 para top(01)
Mensaje D / 00:00:00:000 / registry(05) / 00:00:00:000 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:000 para Root(00)
Mensaje * / 00:00:00:000 / Root(00) para top(01)
Mensaje * / 00:00:00:000 / top(01) para nodegen(03)
Mensaje Y / 00:00:00:000 / nodegen(03) / aout /      1.00000 para top(01)
Mensaje D / 00:00:00:000 / nodegen(03) / 00:00:00:000 para top(01)
Mensaje X / 00:00:00:000 / top(01) / gen /      1.00000 para registry(05)
Mensaje D / 00:00:00:000 / registry(05) / 00:00:00:000 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:000 para Root(00)
Mensaje * / 00:00:00:000 / Root(00) para top(01)
Mensaje * / 00:00:00:000 / top(01) para nodegen(03)
Mensaje Y / 00:00:00:000 / nodegen(03) / aout /      1.00000 para top(01)
Mensaje D / 00:00:00:000 / nodegen(03) / 00:00:01:000 para top(01)
Mensaje X / 00:00:00:000 / top(01) / gen /      1.00000 para registry(05)
Mensaje D / 00:00:00:000 / registry(05) / 00:00:00:000 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:000 para Root(00)
Mensaje * / 00:00:00:000 / Root(00) para top(01)
Mensaje * / 00:00:00:000 / top(01) para deque(04)
Mensaje Y / 00:00:00:000 / deque(04) / aout /      1.00000 para top(01)
Mensaje D / 00:00:00:000 / deque(04) / 00:00:01:000 para top(01)
Mensaje X / 00:00:00:000 / top(01) / deq /      1.00000 para registry(05)
Mensaje D / 00:00:00:000 / registry(05) / 00:00:00:000 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:000 para Root(00)
```

Figure 7: showing the log file for NodeManager

The mobility of nodes within the network is showed on the console by printing a graphical representation of the position of nodes. A screen capture of the console is shown below.

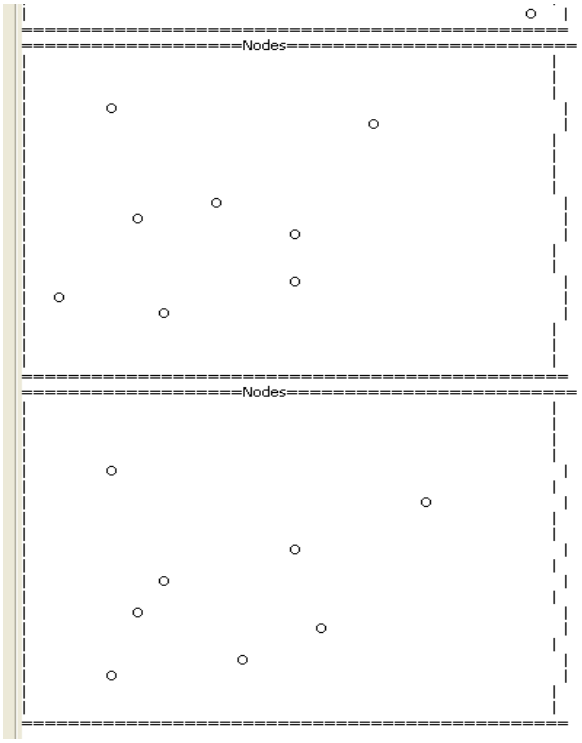

Figure 8: Showing the screen captures of the console (The nodes can be sen to change positions)

The NetStatistics model collects network statistics based on the number of nodes and links in the network. The network reliability of the network is calculated basically as the ratio of the number of links to the number of possible links given the number of nodes in the network. The idea is that if there are n nodes in the network the possible number of nodes in a full mesh mode is n*(n-1)/2. The reliability is the number of available links/ number of possible links. The output of the NetStat model are presented below.

```
00:00:00:000 rel 0
00:00:01:000 rel 0
00:00:02:000 rel 1
00:00:03:000 rel 0.166667
00:00:04:001 rel 0.5
00:00:05:006 rel 0.3
00:00:06:010 rel 0.2
```

Figure 9 showing the output of the netstat model

The network was then built up by coupling nodemanager with the Netstats model as shown by the figure below

```
[top]
components : nodmanager stats@NetStats
out : rel
Link : aout@stats rel
Link : nout@nodmanager nin@stats
Link : lout@nodmanager lin@stats

[nodmanager]
components :  nodep@NodePositions registry@Registry
out : nout lout

Link : out@nodep pos@registry
Link : aout@registry nout
Link : lout@registry lout
```

Figure 10: coupled model defining the network

The network was simulated without an event file it does not have any imputs below are the simulation results:

```
00:00:00:000 rel 0
00:00:00:500 rel 0.25
00:00:01:000 rel 0.392857
00:00:01:500 rel 0.25
00:00:02:000 rel 0.25
00:00:02:500 rel 0.25
00:00:03:000 rel 0.25
00:00:03:500 rel 0.321429
00:00:04:000 rel 0.25
00:00:04:500 rel 0.25
00:00:05:000 rel 0.25
00:00:05:500 rel 0.25
00:00:06:000 rel 0.25
00:00:06:500 rel 0.321429
00:00:07:000 rel 0.321429
00:00:07:500 rel 0.321429
00:00:08:000 rel 0.321429
```

Figure 11: results of the network, presenting the reliablity of the network

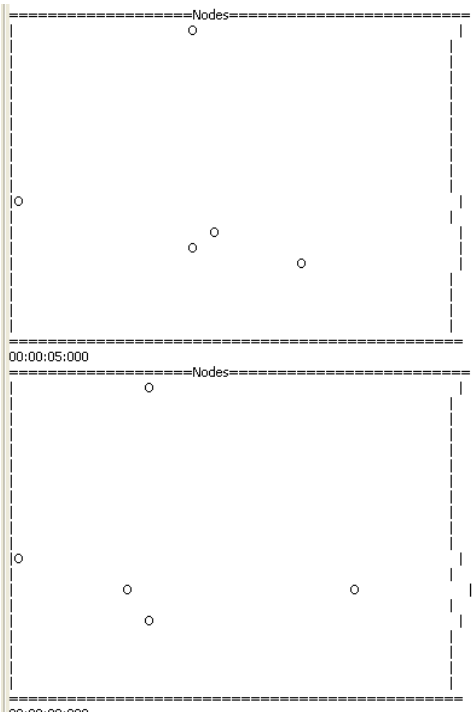

Figure 12 : showing the screen capture of the network nodes as time passes

conclusuion

Different results were obtained by varying the network defaults such as maximum number of nodes, limits of areas, initial number of nodes. The model operated satisfactorily as expected, although given more time the model could be refined and made to better approximate adhoc networks. It is a good starting poit for anyone who is ineterested to model mobile networks. More network parameters coule also be added to the statistics collectiong model to get desired results.