Analyzing a Renaming Algorithm for Mobile Ad Hoc Networks

Sharlita Stevenson
Department of Computer Science
Wayne State University
Detroit, MI 48202, U.S.A
ad3874@wayne.edu

ABSTRACT

This paper studies an algorithm for the "renaming" problem in mobile ad hoc networks, a problem that has not yet drawn much research attention. A Mobile ad hoc network(MANETs) is a collaborative group of mobile nodes that incorporates wireless communication within the network. In the renaming problem, each node begins with a unique id drawn from a large space of possible ids and each node must choose a new unique id drawn from a significantly smaller space of ids. The underlying idea behind a renaming algorithm is to take a long length id and reduce it to a shorter one. A renaming algorithm by Attiya et al.was simulated using the TAMUSim simulator and its behavior. The ultimate focus of this research was to come up with efficient new algorithms for doing renaming in mobile ad hoc networks.

1. INTRODUCTION

The renaming problem in mobile ad hoc networks is one that has not yet stimulated much research focus. The purpose of renaming is to ensure that every node in the network will have a *short* and *distinct(* i.e. no two nodes will have the same identification) id. An advantage of using shorter names is that it reduces the complexity of the messages. According to Attiya et al. 1990, using shorter names as processor identifiers in messages results in shorter messages and hence smaller message complexity. In particular, replacing names from an unbounded domain by bounded-length names lets one to effectively bound the message complexity of algorithms.

Vaidya's paper provides an application for renaming in mobile ad hoc networks. Vaidya's paper describes several mechanisms for duplicate address detection; perhaps using these same methods for duplicate message detection in mobile ad hoc networks would be very beneficial. The reasoning behind examining this problem is to seek a more efficient way of sending messages in the network, by conserving time and producing faster communication between nodes.

The rest of the paper is organized as follows. Section II gives a brief description of related work. Section III contains system model and assumptions . Section IV gives definitions of key concepts. Section V explains the algorithm. Section VI presents the simulator overview. Section VII gives details of the simulation results. Section VIII contains the conclusion.

2. RELATED WORK

This problem has been studied extensively in traditional wired, static networks (Attiya et al. 1990). In Attiya et al. three algorithms are presented; the **simple uniqueness algorithm**(requires a namespace of N=(n-t/2)(t+1), where N represents the size of the network, n represents the number of nodes in the network, and t represents the maximum number of faulty nodes in the network), a more **complex version of the simple uniqueness algorithm**(has namespace of {1,…,n+t}) and an **order- preserving algorithm**(requires N >= 2^t(n-t+1) –1). One application in mobile ad hoc networks for solutions to this problem, identified by Vaidya (2002), is the automatic dynamic assignment of IP addresses to nodes --- in a mobile ad hoc network it is not usually feasible to depend on a primary server to assist with this process and MAC addresses or serial numbers are too long for some routing protocols to handle. Vaidya (2002) weakens the problem statement by allowing the new ids (the IP addresses in his application) to not be unique with some small probability; he then proposes a solution that ensures that, even if duplicate addresses are chosen, all packets sent to the same address will be delivered to the same node (which is one of the nodes with that address).

3. MANET System Model and Assumptions

A mobile ad hoc network(MANET) is a autonomous group of mobile nodes forming a network that is completely independent of any preexisting infrastructure, which incorporates wireless communication links between every node in the network. The mobile ad hoc network here was modeled as different arrangements of n nodes. Assumptions on the mobile nodes are:

1. The nodes are mobile implying that when a communication link is dropped between two nodes(i.e, an edge is deleted), the node which is partitioned from the network can be reconnected at a different location within network.

2. Each node is aware of who their neighbors are by communication links(i.e, bi-directed edges) between themselves and the adjacent node.

3. The message(s) on a failed communication link are automatically assumed to have been delivered to its destination, in other words, there are no lost messages.

4. DEFINITIONS and KEY CONCEPTS

The components/variables of the algorithm can be informally defined as follows. Each node utilizes the following:

- Old vector(V_oldInfo = V') – holds old information
- New vector(V_newInfo = V) – holds new information
- Message vector(receivedMsgs) – keeps track of messages a node has received
- System Id(sys_id) – system id of a node
- Old Id(old_id) – original id of a node
- New Id(new_id) – new id assigned to a node as a result of renaming
- Total nodes(int n) – total number of nodes in the network
- Counter(int c) – number of nodes claiming to have the same set V
- Maximum number of faulty nodes(int t) – upper bound of faulty nodes in network.

5. ALGORITHM

This work studies the simple uniqueness algorithm of Attiya et al. This algorithm doesn't have a way of broadcasting a message throughout the network, it's only capable of sending a message by a direct link. In order to adapt the simple uniqueness algorithm to the MANET environment, we need a mechanism that will propagate a message to every node within the network. We use a flooding technique as a means of broadcasting a message. The algorithm incorporated with flooding generates a renaming message which is sent to every node in the network. Once a node gets the message, it checks to see whether or not it has received the message before. If Here node i is used to denote a node 0 through node n-1. Before the algorithm is executed by node i, every node in the network has an initial renaming message, which contains the sender's id and message number. When the code is executed and node i receives a message, it checks to see if that message has been sent before. If it has been sent before, the message is simply ignored. However, if the message has not been sent before, node i updates its receivedMsgs with the sender's id and message number, and forwards the message to its neighbors, who in turn forwards the message to their neighbors and so on. The next section of the algorithm determines whether or not a node has chosen a new id. Next, we check to see if node i has chosen a new id. If node i has chosen a new id, an update message is sent to its neighbors, who adds the new message to their receivedMsgs and circulates the new message to their neighbors. If node i has not chosen a new id, it falls into one of the three cases:

1. V' is a subset of V
2. V' contains an element not in V
3. V' is equal to V

Here we explain details of each case.

Case 1: V' is a subset of V if and only if V' contains elements which are in V. In the code, this case denotes that no new information is found, so nothing is done.

Case 2: V' contains an element that is not in V if V' contains different elements than that of V, which means V' and V are two different sets. In the code, this case denotes that there is new information, so update V' with the new informations, forward the new information to neighbors, and update receivedMsgs.

Case 3: V' is equal to V if and only if V' contains every element in V and V contains every element in V'. This case denotes that node i's c >= (n-t), which means the number of nodes that have claimed the same set V as itself. This implies that node i has no more information to forward to its neighbors and V is "stable"(Attiya, 1990). When node i has c = n-t identical messages (or sets V), then node i should take some action; thus, taking action of chosing a new id.

With flooding every node in the network receives data and forwards that data to all of its adjacent neighbors. Here, each node used the flooding approach as a way to broadcast(i.e, send a message) its newly received information which when received by the other nodes(i.e, neighbors) is kept track of. Each node maintains old (with vector V_oldInfo) and new information (with vector V_newInfo) of every other node in the network. It also maintains an integer counter(c as above). Each message is associated with a message id and counter which denotes who sent the message and the number of times it was sent. When a node has received the message, it informs(i.e, broadcasts a message) its neighbors of any new information; performing such an operation prevents a node from receiving duplicate messages.

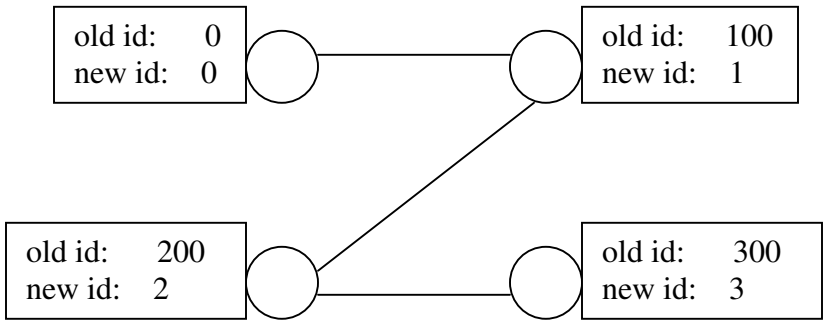
6. SIMULATOR OVERVIEW

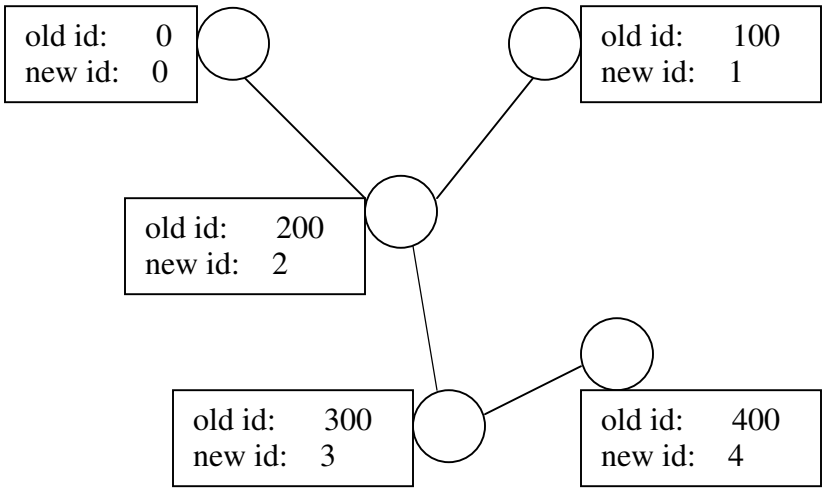
TAMUSim is a general algorithm-level simulator for MANET, presently that can be used to simulate simple algorithms for mobile ad hoc networks. It is written in the Java computer language and is used to automate various algorithms and network topologies. It is designed to help understand qualitative behavior, not performance behavior, of an algorithm. It assists the user in developing correctness proofs and automation of possible counter-examples. It also provides a graphical user interface(GUI) where the user can interact with the simulator. The renaming algorithm was implemented as a module that interacted with the simulator by giving it specific instructions to execute the module.

7. SIMULATION RESULTS

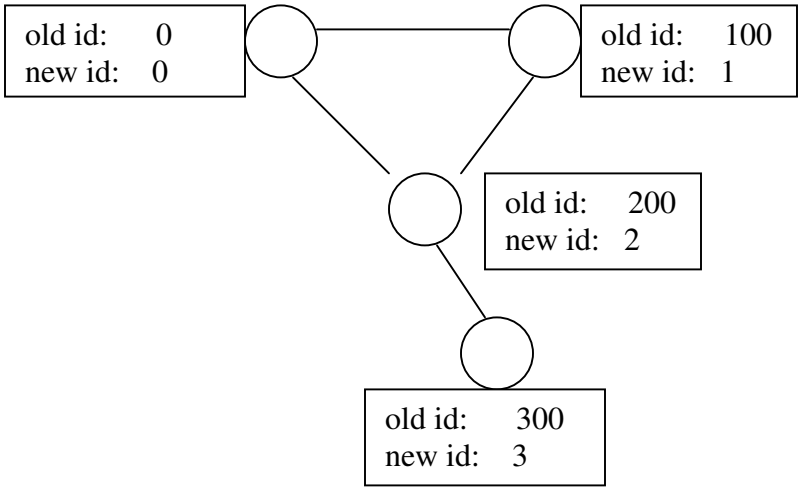
The simulation environment consisted of at most four nodes. The decision to use this limited number of nodes derived from the correlation between the number of nodes used and the increased accumulation of messages. The simulations were run on the

TAMUSim simulator using different network arrangements. Next are graphs displaying the nodes with their initial configuration and their new ids.





Note: using a graph with more than four nodes generates a large amount of messages.



In the graphs above, the lines between each node represents a bi-directional edge (i.e, communication link). The new ids are distinct from one another. According to Attiya et al. compared to longer ids, the use of shorter ids decreases the complexity of a message. When a message is sent by a node, only its adjacent neighbors receive the message. For example, if node 0 sends a message, the message is received by node 1 and node 2, then node 2 can forward the message to node 3.
During the simulation executions a restricted amount of nodes were used based upon the correlation between the total nodes used and the size of messages generated. When more than four nodes were used in the simulation, the list of messages became extremely lengthy.

8. CONCLUSION

This paper has examined the renaming algorithm along with simulation implementations for a mobile ad hoc network. The main idea behind the algorithm was to let each node begin with a unique id drawn from a large space of possible ids and eventually, each node had to choose a new unique id drawn from a significantly smaller space of ids. When running the simulations it became apparent that there was a dependency between the size of the network and total number of messages. The flooding mechanism worked as a means of broadcasting, but was not efficient in the mobile ad hoc network environment.

Being that the renaming algorithm has not received very much research attention in MANETs there could be many potential and beneficial areas that can be derived from it. The idea of implementing node mobility was attempted as an addition to the renaming algorithm, however, due to the complex underlying details, it left for of future work. The basic notion of node mobility is that when a node is disconnected from a network(i.e. a partition has occurred), it will take any necessary actions when this situation happens.

The problem of renumbering the nodes also becomes an issue when a partition is detected in a network. The inherent complexity of the renaming problem and the methods used in obtaining the algorithms are basic and are useful both in their own right and in attacking other problems in asynchronous message passing environment(Attiya, 1990). .
One possible enhancement for the TAMUSim simulator is to implement dynamic automation for simulations, instead of manual intervention(i.e, using mouse or keyboard).

9. ACKNOWLEDGEMENTS

The research reported here was conducted at Texas A&M University and supported by the Distributed Mentor Project(DMP). I would like to thank Dr. Jennifer Welch, Yu Chen, and Vijay Balasubramanian for their tremendous support, helpful discussions, and comments. This project would not have been the meaningful experience that it was without their guidance and assistance.

10. REFERENCES

Nitin Vaidya, "Weak Duplicate Address Detection in Mobile Ad Hoc Networks," *Proceedings ACM MOBIHOC*, 2002. pp. 206-216.

Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Ruediger Reischuk, "Renaming in an Asynchronous Environment, " *Journal of the ACM*, vol. 37, no. 3, July 1990, pp. 524-548.

APPENDIX

The following code(RenameNode.java) is an implementation of simple uniqueness algorithm by Attiya et al. The other .java classes work with RenameNode.java.

```
//RenameNode.java
package Modules.RenameStuff;
import Sim_internals.graph.*;
import Sim_internals.message.*;
import java.util.*;


public class RenameNode extends GenNode
{

    private Vector V_newInfo;      /* vector with new information */
    private Vector V_oldInfo;      /* vector with old information */
    private Vector receivedMsgs;   /* keeps track of messages received */
    private int sys_id;            /* system id for node */
    private int old_id;            /* original id of node */
    private int new_id;            /* new id of node */
    private MsgID msg_id;          /* message object to hold id and counter */
    private RenamingMsg m;         /* object of RenamingMsg */
    private int msg_num;           /* message counter */
    private boolean isUnion;       /* flag indicating a union */
    private boolean isSubSet;      /* flag indicating a subset */
```

```
private boolean isEqual;       /* flag indicating equality */
private boolean bEcho;         /* flag to check echophase */
private int v;                 /* size of old vector(V_oldInfo) */
private int r;                 /* # of elements(i.e. id's) in V_oldInfo
                                   that are < original id */
private int n;                 /* # of nodes */
private int t;                 /* # of faulty nodes( deleted nodes ) */
private int c;                 /*counter for # of nodes that have
                               claimed having the same set V as itself*/


/****************
*  constructor  *
****************/
public RenameNode( int sys_id )
{

    super( sys_id );
    V_oldInfo = new Vector();
    V_newInfo = new Vector();
    receivedMsgs = new Vector();

    /** initializing **/

    this.sys_id = sys_id;
    old_id = sys_id * 100;      /* assigns id in larger range */;
    msg_num = 0;
    bEcho = false;

    Integer i = new Integer(old_id);
    V_oldInfo.addElement(i);

   n = genGraph.getSize();     /*gets # of nodes in graph */
    msg_id = new MsgID( sys_id, msg_num );
   PrintVector();
   new_id = -1;
   c = 1;
    t = 0;
   //t = (n/2);

}


/*************************************
*  This function begans the broadcast  *
*   (i.e, sends V to everyone)         *
*************************************/
public void startAlg()
{
    System.out.println("*****************************************");
    System.out.println("Flooding message has began.");

    msg_id = new MsgID( sys_id, msg_num );
   Vector vector = new Vector(V_oldInfo);
    RenamingMsg msg = new RenamingMsg(sys_id, -1, vector, msg_id, "In StartAlg");

    msg.PrintVector();

    broadcastMessage( msg);
    receivedMsgs.add(msg_id);
    msg_num++;        /* updating message counter */

}



 /************************************************************
 * instructions for how a node should react for different cases;*
 * depending on the case, there are directions to carry out the *
 * next step.                                                   *
 ************************************************************/
```

```
public void receiveMessage( Object o )
{
   /* check o to see what type of msg it is */

   if( o instanceof LinkDownMessage )
   {
        System.out.println("This is a linkdownmsg, edge is deleted.");
        //nodePartition(o);
       return;
   }

   if( o instanceof LinkUpMessage )
   {
        System.out.println("This is a linkupmsg, edge is added.");
        //nodePartition(o);
       return;
   }

   if( !(o instanceof RenamingMsg))   return;

   System.out.println("");
   System.out.println("");
   System.out.println("***********************************************");
   System.out.println("Node " + sys_id +" receives a renaming message.");


   /* cast to the appropriate type and add new info to vector */
   ((RenamingMsg)(o)).PrintVector();
   V_newInfo.removeAllElements(); /* clearing the vector of any garbage */
   for (int y = 0; y < (((RenamingMsg)( o )).getNames()).size(); y++)
   {
       V_newInfo.add( (((RenamingMsg)( o )).getNames()).elementAt(y) );
   }


    /*******************************************
   *  check if node hasn't received a msg with  *
    *     a particular msg_id                    *
    *******************************************/

    int senderID= (((RenamingMsg) o).getMsgID()).getOriginator();
    int seqnum = (((RenamingMsg) o).getMsgID()).getNumber();
    MsgID messageID = new MsgID (senderID, seqnum);

    System.out.println("Step 1: Check whether the msg has been received before:");

    /*************************
   * extract the msg_id here *
    *************************/
   System.out.print("Element in receivedMsgs: ");
   for( int s=0; s < receivedMsgs.size(); s++ )
   {
       System.out.print("["+
       ((MsgID) receivedMsgs.elementAt(s)).getOriginator() + "," +
       ((MsgID) receivedMsgs.elementAt(s)).getNumber()+"]");

       if( messageID.equals((MsgID)receivedMsgs.elementAt(s)))
        {
          System.out.println("");
           System.out.println("Do nothing, msg_id is in receivedMsgs.");
          return;
        }

   }
   System.out.println("");
    System.out.println("This message has not been received before.");
    Vector vector = new Vector(((RenamingMsg) o).getNames());
    broadcastMessage( new RenamingMsg( sys_id, -1, vector,
                                       messageID, " Relay the message"));
    receivedMsgs.add( messageID ); /* updating vector */
```

```
        System.out.println("The message has been relayed.");


        /*************************************
        *  checking to see if new id have been  *
        *  chosen.                              *
        *************************************/
        System.out.println("");

        if( !(new_id == -1) )
        {
            /************************
            *  echophase:           *
            *   have chosen new_id  *
            ************************/

            System.out.println("Step2: Echo Phase, a message with vector is broadcast");
            Vector vector1 = new Vector(V_oldInfo);
            msg_id = new MsgID(sys_id, msg_num);
            RenamingMsg msg = new RenamingMsg(sys_id, -1, vector1,
                              msg_id, " After Union");
            msg.PrintVector();
            broadcastMessage( msg);
            receivedMsgs.add(msg_id);  /* updating vector */
            msg_num++;
            System.out.println("msg_num = " +msg_num);
        }

        else
        {
            System.out.println("Step2: This node hasn't yet chosen new_id, check the three
cases:");
            PrintVector();
            /****************************
              *   Check Different Cases   *
              ****************************/

            /*************************************
              * Note: V = V_newInfo & V' = V_oldInfo *
              *************************************/

            /*********************************
              *  Case 1: V' is a subset of V  *
             *********************************/

            /** do nothing **/

            isSubSet = (SubSetof(V_newInfo, V_oldInfo)) && (!Equal(V_newInfo,V_oldInfo));
            if( isSubSet )   /* if true */
                System.out.println( "Case 1: Do Nothing." );

             else
             {
             /*****************************
             *   Case 2: V <-- (V u V')   *
             *****************************/

              /** V' has a value not in V **/

               isUnion = Union( V_newInfo, V_oldInfo );
               if(!isUnion )
               {
                    System.out.println("Case 2: isUnion");
                    for(int z=0; z < V_newInfo.size(); z++)
                   {

                       if(!( V_oldInfo.contains(V_newInfo.elementAt(z))))
                          V_oldInfo.add( V_newInfo.elementAt(z));
                   }
                   System.out.println("The vector is updated:");
                   PrintVector();
```

```
                    Vector vector1 = new Vector(V_oldInfo);
                    msg_id = new MsgID(sys_id, msg_num);
                    RenamingMsg msg = new RenamingMsg(sys_id, -1, vector1, msg_id, " After
Union");
                    System.out.println("A message with the updated vector is broadcast:");
                   msg.PrintVector();
                    broadcastMessage( msg);
                    receivedMsgs.add(msg_id);
                   msg_num++;
                    System.out.println("msg_num = " +msg_num);
                    c = 1;
                }

               else
               {
              /*****************************
              *       Case 3: V'= V       *
              *****************************/

                isEqual = Equal( V_newInfo, V_oldInfo );
                 if( isEqual )
                 {
                     System.out.println("Case 3: isEqual");
                    c = c + 1;
                     System.out.println("c = " +c);
                     if( c >= (n - t) )
                    {
                          bEcho = true;
                           v = V_oldInfo.size();   /* v <- |V| -- (use size method) */
                           r = 0;    /* r <- # of elements < original id */

                           Integer old_id_obj = new Integer( old_id );

                           /***********************************
                          *  Checking for the rank(i.e. "r") *
                           *    of a specific id  *           *
                           ***********************************/
                          for( int x=0; x < v; x++ )
                          {

                               if( ((Integer)V_oldInfo.elementAt(x)).compareTo(old_id_obj)
< 0 )
                                 r++;
                           }

                           /*****************************************
                          *  new_id <- Mapping(v,r) -- (give an        *
                           *  integer in range 1 to (n - t/2)(t + 1) )  *
                           *****************************************/
                             new_id = Mapping( v, r );
                            System.out.println("v=" + v + ", r=" + r + " new id = "+new_id);


                     }/** end if **/

                 }/** end if **/

                }/**end 2nd else **/

              }/**end 1st else **/

        }/** end if **/

 }/** end ReceiveMessage() **/

/*****************************************
* This function is a one-to-one function     *
* that maps the pair( v, r ) to the new_id.  *
*****************************************/
public int Mapping( int pv, int pr )
```

```
{

    int sum = 0;

    /** f(u,v) = (n-t) + (n-t+1) +.....+ (v-1) + r **/

    for( int m = n-t; m <= pv-1; m++ )
    {
        sum = sum + m;

    }
    sum = sum + pr;
    return sum;
}


/*****************************************
* This function returns true if and only *
* if every element of v1 is in v2.       *
*                                        *
*****************************************/
public boolean SubSetof( Vector v1, Vector v2 )
{

    int i, j;
    Integer a;
    Integer b;
    boolean found;


    for( i = 0; i < v1.size(); i++ )
    {
        found = false;
        for( j = 0; j < v2.size(); j++ )
        {
            if((v1.elementAt(i)).equals(v2.elementAt(j)))
                found = true;
        }

        if( found == false )    /* v2 isn't a subset of v2 */
            return false;
    }
    return true;
}

/*****************************************
* This function returns true if v2 has info *
* different from v1.                       *
*                                          *
*****************************************/
public boolean Union( Vector v1, Vector v2 )
{

   boolean diff_elem;
   diff_elem = SubSetof( v1, v2 );
   return diff_elem;
}

/*****************************************
* This function returns true if both vectors, *
* v1 and v2 are equal.                        *
*                                             *
*****************************************/
public boolean Equal( Vector v1, Vector v2 )
{
   return  (SubSetof( v1, v2 ) && SubSetof(v2,v1));
}

/*****************************************
* A simple function that prints the contents of *
* V_oldInfo and V_newInfo for each node.        *
```

```
*********************************************/
public void PrintVector()
{
        System.out.println("------------------------- Node State ----------------");
        System.out.println("The state of vectors on node "+ sys_id +" with old_id " +
old_id);
       System.out.print("V_oldInfo:");
       for(int i=0; i<V_oldInfo.size(); i++)
       {
            V_oldInfo.elementAt(i);
            System.out.print(V_oldInfo.elementAt(i)+",");

        }
       System.out.println("");
       System.out.print("V_newInfo:");
        for(int k=0; k<V_newInfo.size(); k++)
       {
            V_newInfo.elementAt(k);
            System.out.print(V_newInfo.elementAt(k)+",");

        }
       System.out.println("");
       System.out.println("-------------------------------------------------------");
}


}/** end of class RenameNode **/
```

```
//RenamingMsg.java
//This class creates a renaming message

package Modules.RenameStuff;

import java.util.*;

public class RenamingMsg
{

    private Vector names;   /* holds the data( the names ) */
    private MsgID msg_id;   /* object of type MsgID */
    private String info;
    private int sid;        /* sender id */

    public RenamingMsg( int sender, int receiver, Vector ids, MsgID m_id, String
information )
    {
        sid = sender;
        info = information;
        msg_id = new MsgID(m_id.getOriginator(), m_id.getNumber());
        names = ids;
    }

    /*****************************************
    * A simple print function that outputs the *
    * current state of the vector( names ).    *
    *****************************************/
    public void PrintVector()
    {
        System.out.println("---------------------- Message State --------------------------
");

       System.out.println("msg ("+msg_id.getOriginator()+","+msg_id.getNumber()+ ")" );
       System.out.print("The state of vectors on msg:");
        for(int i=0; i<names.size(); i++)
       {
            System.out.print(names.elementAt(i) +",");
        }
       System.out.println("");
       System.out.println("---------------------------------------------------------------
");
```

```
        System.out.println("");
    }

    /*****************************************************
    * getMsgID: helper function that gets the message id *
    * of the message being sent.                         *
    *****************************************************/
    public MsgID getMsgID()
    {
        return msg_id;

    }

    /*****************************************************
    * getNames: helper function that gets the names(i.e. *
    * ids) of the nodes.                                 *
    *****************************************************/
    public Vector getNames()
    {
         return names;
    }

    /*************************************************
    * Simple print function to display information is *
    * pop-up window of gui screen.                   *
    *************************************************/
    public String toString()
    {
        return new String("Renaming Message:  vector "+ names + " Note: " + info);

    }

} /* end of class RenamingMsg */
```

```
// MsgID.java
//This class combines all the elements of a message( i.e, sender id, message number)

package Modules.RenameStuff;

public class MsgID
{
    private int originator;             /* node that forwarded the message */
    private int number;                 /* message number */

     /* constructor */
     public MsgID( int oid, int num )
     {
         originator = oid;
         number = num;

     }

    /* checks if argument MsgID object has equal value to this one */

    public boolean equals(MsgID mi)
    {

        return ((mi.getOriginator() == originator) && (mi.getNumber() == number));
    }

    public int getOriginator()
    {
        return originator;
    }

    public int getNumber()
    {
        return number;
    }
```

```
} /* end of class MsgID */
```

```
// Rename.java
//This is the main class that interacts with the algorithm

import Sim_internals.graph.*;

import Modules.RenameStuff.*;

public class Rename extends Algorithm
{
    // To create a node, call constructor specific to the
    // Renaming algorithm.

    public GenNode createNode(int id)
    {
        return new RenameNode(id);
    }

    /* To initialize the algorithm, call initializer specific
      to the Renaming algorithm. */

    public void initializeSimulation()
    {
        int i;

        for( i = 0; i <genGraph.getSize(); i++ )  /* loop until? */
        {
            // Integer j = new Integer( i );
            GenNode source = genGraph.getNode( i );
            ((RenameNode) source).startAlg();
        }
    }

} /* end of class Rename */
```