# CS 6390 Spring 12 Programming Project

NOTE: there are always typos/omissions/etc., please help me find them. Also, please read it ASAP and start working on the code. It may take longer than you think.

NOTE: This is INDIVIDUAL work. People sharing code will be sent to the Dean of Students for disciplinary action.

## System/Language requirements

Your program can be written in any language that you wish, the only restriction is that it must run in the Unix machines here on campus (no, not on your laptop). Most, of course, prefer Java and C++, but I have had projects turned in Perl and in Pascal (no kidding).

When you turn in your project, you can zip together all the source files. In addition, you should include a "README" file that contains:

1. The name of the Unix machine you used to run your project
2. Compilation instructions for your project (PRECISE compilation instructions, including the compiler name and any flags as arguments to the compiler)
3. Obviously your name and ID :)

We will compile your source code and run it against our test cases.

The due date is to be determined, but it will be around one week before the final grades are due.

## Overview

### Processes and arguments

We will simulate a very simple network by having a process correspond to a node in the network, and files correspond to channels in the network.

We will have at most 10 nodes in the network, nodes 0, 1, 2, . . . , 9

Each process is going to be given the following arguments

1. id of this node (i.e., a number from 0 to 9)
2. the destination id of a process to which the node should send data
3. a string of arbitrary text that the node will send to the destination
4. the time at which it should transfer the string to the destination

We will have a **single program** node.c (or whatever extension you want to use, node.cc, node.java, whatever) which has the code for a node. Since we have multiple nodes, we

will run the same program multiple times, in **parallel**. The only difference between each of the copies of the program running in parallel is the arguments to the program.

For example, assume I have two programs, A and B, and I want to run them at the same time. At the Unix prompt >, I would type the following

> A &

> B &

By typing the & we are putting the program in the "background", i.e., the program runs in the background and you can keep typing things at the terminal. Therefore, A and B are running in parallel at the same time.

Again, let “node” be your compiled program that represents a node. The arguments of the program are as follows

node 3 5 "this is a message" 25 &

The following would execute the program node, and the first argument is the id of the node (3) the second is the destination for this node (5) the third is the message string "this is a message", and the fourth is the time at which the node begins attempting to transmit the string to the destination (i.e. after 25 seconds of execution).

For example, assume I have a network with three nodes, 0 , 1, 2, and I want node 0 to send a string "this is a message from 0" to node 2 (starting at time 10), and node 1 to send a message "this is a message from 1" to node 2 (starting at time 20). Then I would execute the following commands at the Unix prompt > (your prompt may be different)

> node 0 2 "this is a message from 0" 10 &

> node 1 2 "this is a message from 1" 20 &

> node 2 2 &

This will run three copies of node in the background, the only difference between them are the arguments each one has.

For node 2, since the "destination" is 2 itself, this means 2 should not send a string to anyone.

There will be a single additional process, called **controller**. The controller is needed because of the way that we handle I/O, as will be discussed below.

All processes must be started at the same time. Since the above takes a while to type, you can write a small shell script to quickly spawn all the above processes. A sample shell script will be provided.

## Communication Model and Files

We will model a wireless network (well, it is of course a far cry from a wireless network but nonetheless . . . ). I will assume that when a node sends a message all of its neighbors are able to hear the message (since it is a broadcast medium).

Each node with ID X will open to text files: toX.txt and fromX.txt. When X sends a message it **appends** it to fromX.txt (it does not overwrite any previous content of the file). Hence, at the end of the execution, fromX.txt contains all the messages sent by X.

Messages are separated by a newline (to make viewing/parsing easier)

A node is not aware of who its neighbors are. The "controller" is the process that is aware of the entire topology of the system. In real life the controller would not exist, but since we are using files rather than a real wireless network, the controller will act like the wireless network.

If the controller knows there is a link from X to Y (i.e., X and Y are neighbors, WARNING, links can be one-directional) then when X appends a message to fromX.txt, the controller will read fromX.txt, notice the new message, and copy the message to the file toY.txt.

Thus, node X receives in the file toX.txt a copy of every message sent by any of its neighbors.

How does the controller know who is a neighbor of whom? Before the execution, you will create by hand a file called topology.txt. This file contains multiple lines, each line is of the following form.

time status node1 node2

Each line corresponds to the "activation" or "deactivation" of the one-directional link (node1 → node2). For example, if 10 seconds after the system begins to execute node X and Y become close to each other (and become bidirectional neighbors) then the topology.txt file will have the link

10 UP X Y
10 UP Y X

Where X and Y are numbers in the range 0 .. 9 (i.e. node ID's)

If at time 20 X and Y become far away from each other then topology.txt will have the file

20 DOWN X Y
20 DOWN Y X

You can assume the entries in topology.txt are sorted by time. Also, it is possible that the link from X to Y remains up while the link from Y to X goes down.

**ONCE EVERY SECOND** the controller will read **ALL** fromX.txt files (for every X 0 .. 9) to check if there are new messages sent, and if so the messages are forwarded to the appropriate toX.txt files.

**ONCE EVERY SECOND** each node X will read the toX.txt file to see if there are any new messages to process.

# Routing

Routing of messages from one node to another will be performed using the AODV protocol. I will not include all details here since you know how AODV behaves, but I will provide some of the highlights.

There will be three types of messages:

RREQ

RRPLY

DATA

When each of these messages is written into the file fromX.txt by node X, they are preceded by the node id of the next-hop node that should receive this message.

E.g., consider DATA messages, they are of the form

<nxthop> <fromnbr> DATA <srcnode> <dstnode> <string>

One example of a data message would be

2 1 DATA 0 3 This is a message

This message is being sent from node 1 to node 2 (i.e. along the link 1 → 2), and it is a data message which originated at node 0 and whose ultimate destination is node 3. The data in the message is “This is a message”.

## Sending the String to the Destination

After the appropriate number of seconds (i.e. the last argument of the program) have elapsed since the beginning of the execution, then the node attempts to send the string to the destination.

If in its routing table it has a route to the destination, then it uses this route to send the string (i.e. it puts the string in the data message and sends its to the next hop).

If no router is found in the routing table, then a new RREQ is sent out looking for the destination.

The node waits 30 seconds for the RRPLY. If no RRPLY is received in 30 seconds a new RREQ is sent.

## RREQ Processing

The format of a RREQ message is as follows

<nxthop>  <fromnbr> RREQ <srcnode> <dstnode> <bdcastid> <srcseq#> <dstseq#> <hopcount>

All these fields should be self-explanatory if you understand AODV. The only fields not related to AODV are the <nxthop> and <fromnbr> fields, which are the same as in a DATA message (they are basically are the “link-level” destination and source of the message).

<nxthop> however corresponds to all neighbors of the node. Hence, we use “*” (star) for the next-hop node in this case, indicating that the message is a broadcast to all the neighbors.

The only difference in behavior that I would like that is different from the standard is as follows. We know that RREQ’s are forwarded only once. In the standard, if you have seen a RREQ before, you drop it even if it offers a better route. In our program, if you have seen the same RREQ before, but if it offers a shorter hop count, then, we DON’t forwarded it, but we DO update the routing tables to point towards the shorter path.

It is possible for a RREQ to be terminated by an early RRPLY, as discussed in class. If this is the case, then a gratuitous RRPLY is sent to the destination.

Obviously, when the destination receives the RREQ, it sends back a RRPLY. If it sees the same RREQ again (same bdcast id) you don’t reply to it, but if it offers a shorter path to the source you update your routing tables accordingly.

When you set up a new routing table entry, the value of active_route_timeout will be 20 seconds. Every time data is received that uses that routing table entry, the expiration time of the route is set to the current time plus 20 seconds more.

## RRPLY Processing

The destination returns a RRPLY as discussed above, also intermediate nodes may return an early route reply.

<nxthop>  <fromnbr> RRPLY <srcnode> <dstnode> <dstseq#> <hopcount>

The values are as expected. Notice there is no lifetime. We will only use the value of active_route_timeout discussed above.

## Processing a Data Message

When a data message is received, it is forwarded to the next node according to the routing table. If there is no next hop information for this destination, the message is dropped.

In addition, the set of active neighbors corresponding to the message is updated. A node X is in the set of active neighbors of node Y for destination Z if Y has received a data message from X whose destination is Z in the last active_route_timeout seconds.

A routing table entry is active as long as its list of active neighbors is not empty. When the list of active neighbors becomes empty then you remove the routing table entry from the routing table.

If node X receives a data message that has been addressed to it (X is the destination) then X appends the string in the data message to the file Xreceived.txt. I.e., this file will have all the data strings that X received during its execution.

## Maintaining a set of Active Neighbors

Each routing table entry must contain a set of active neighbors. If a neighbor sent a data message that used the routing table entry during the past 20 seconds, then the neighbor must be in the set of active nieghbors for that routing table entry.

I.e., when a neighbor has not sent a data message that uses that routing table entry in the last 20 seconds, then you can remove it from the active neighbor list of that routing table entry.

Again, note that each routing table entry has its own set of active neighbors.

## Link Maintenance

Each node X sends out, every 2 seconds, a “hello” message, of the following form:

* X HELLO <nbr1> <nbr2> . . . <nbrn>

Where <nbr1> <nbr2> . . . <nbrn> is the list of neighbors of the node. If X is not aware of any node, then this list is empty.

AODV works only on bidirectional links. Thus, if X can receive a hello message from Y, but Y does not include X in its list of neighbors, then the link between X and Y is not bidirectional.

If X believes that Y is a neighbor (and the link is bidirectional), and in the last 6 seconds X has not received a HELLO message from Y with X included in the list of Y’s neighbors, then X considers the link (X,Y) to be down.

### Error messages when the link breaks

An error message has to be sent to the active neighbors of the link , which in turn is sent to their active neighbors, etc., until it reaches the source.

I will let you format this message any way you want. You can introduce a new message type if you wish, or you can use a gratuitious RRPLY. This RRPLY should have a sequence number at least one greater than the sequence number of the destination and a hop count of infinity (you can define infinity any way you want, either as -1 or as 11 since we only have 10 processes. The ID of the source can be anything you want, it could be 11 if you wish too.

# Program Skeleton

You can write your “node” program with an overall general structure as follows (of course details have to be added by you). You can figure out the controller on your own.

```
main()

  initialize variables
  open fromX.txt for appending, toX.txt for reading,
       and Xreceived for appending.
  i = 0
  while (i < 100)
     read toX.txt
     process any new received messages
     if it is time to send the data string
            if there is a routing table entry for the destination
               send the data message
            otherwise. Initiate a rreq
     i = i + 1;
     sleep for 1 second.
```

```
end while
close files
end program
```

# General Remarks

**DO NOT RUN YOUR PROGRAM WITHOUT THE SLEEP COMMAND.** Otherwise you would use too much CPU time and administrators are going to get upset with you and with me!

**Notice that your process will finish within 100 seconds (or about) after you started it.**

Note that you have to run multiple processes in the background. The minimum are two processes that are neighbors of each other, of course.

After each "run", you will have to delete the channel and output files by hand (otherwise their contents would be used in the next run, which of course is incorrect).

Also, after each run, **you should always check that you did not leave any unwanted processes running, especially after you log out !!!** To find out which processes you have running, type

ps -ef | grep userid

where userid is your Unix login id. (mine is jcobb). That will give you a list of processes with the process identifier (the process id is the large number after your user id in the listing)

To kill a process type the following

kill -9 processid

I will give you soon a little writeup on Unix (how to compile, etc) and account information. However, you should have enough info by now to start working on the design of the code

Good luck