CS 6390 Fall 2024 Programming Project

There are probably many typos/omissions/mistakes. The sooner you read it and start working on it the sooner we will find and correct mistakes :)

YOUR PROGRAM MUST RUN IN THE UNIX MACHINES ON CAMPUS: I recommend you use csgrads1.utdallas.edu. Note that this machine may be inside of UTD's firewall, so you may have to ssh into pubssh.utdallas.edu, and from there ssh to csgrads1.utdallas.edu

Due: 5 days before grades are due, i.e., December 13 AT 8:00 AM IN THE MORNING !!!! Be mindful of this time !!!

Overview

The network we will simulate will be a network with unidirectional channels (links). Thus, if from node x to node y there is a channel it does not imply that there is a channel also from y to x. We will build a distance-vector-type of routing protocol, and then build a multicast protocol on top of it (and no it is not DVMRP, is more like PIM)

We will simulate a very simple network by having a unix process correspond to a node in the network, and files correspond to channels in the network. Thus, if I have two nodes x and y in the network, and a channel from x to y, x and y will be unix processes running in the background (concurrently) and the channel from x to y will be a unix file.

We will have at most 10 nodes in the network, nodes 0 , 1, 2, . . . , 9. There will also be a special node, a "controller" node. This node does not represent any "real" node, but we need it due to the fact that we will use files to represent channels.

Process names and arguments

All "regular" nodes will have the same code, the filename should be called node.cc (if c++ is used, for example, node.java for java, etc), the controller should be simply called controller.cc (if c++ is used).

If a node is neither a sender nor a receiver of the multicast group, then it will be executed as follows

> node ID duration &

where ">" is the unix prompt (which varies from machine to machine), "node" is the executabe code of the node, ID is from 0 to 9, "duration" is the number of seconds the process should run before it termintates itself, and "&" indicates that the process will be run in the background (freeing up the terminal to do other things). E.g.,

> node 5 100 &

executes node 5 in the background for 100 seconds.

If the node is a sender, it will be executed as follows

> node ID sender string duration &

where sender is simply the string "sender", and string is an arbitrary string of data that the sender will multicast to the receivers. E.g,

```
> node 8 sender "this string will be multicast" 100
```

If a node is a receiver, it will be executed as follows

```
> node ID receiver S duration &
```

where S is the id (0 .. 9) of the sender whose tree the receiver would like to join. E.g.,

```
> node 3 receiver 8 100 &
```

would execute a node whose ID is 3 and is a receiver and wants to receive from node 8, and hence, it will receive the string "this string will be multicast" from the tree of node 8.

The controller is simply executed as follows

```
> controller duration &
```

Channels, Processes, and Files

Assume there is a channel from y to x (y --> x). In this case, we say that:

- y is an incoming neighbor of x
- x is an outgoing neighbor of y

Each node will periodically broadcast a hello message (more details on the hello protocol later below). E.g., when y sends a hello message, x will receive it, and x will add y to its list of known incoming neighbors.

Since there is a channel from y to x and no channel from x to y, then x becomes aware that it has an incoming neighbor y, but y is NOT aware it has an outgoing neighbor x since it cannot receive messages from x.

Note that if there are two channels, y --> x and x --> y, both x and y will learn that the other process is an incoming neighbor of them, but they do not know that the other process is also an outgoing neighbor. E.g, y knows it can receive messages from x but it does not know x can receive messages from y.

Each node x has a single output file, called output_x, where x is from 0 to 9.

Each node x will also have a single incoming file, input_x, where x is from 0 to 9.

The input and output files of a process consist of a sequence of messages, each message will be in a separate line in the file.

Each message sent by a node is heard (received) by all its outgoing neighbors, but a node is not aware of who its outgoing neighbors are. In our simulation, however, how can we implement this? We solve this by introducing a "controller" node. This node will implement the fact that not all nodes can reach all other nodes in one hop. Thus, the controller will have an input file (which you write before the execution of the simulation) that contains the network topology, i.e., the file describes which nodes are neighbors of each other and which ones are not.

Thus, if when node y sends a message, it appends it to its output file output_y

The controller will read file output_y, and check the topology of the network. If x is an outgoing neighbor of y, then it will append this message to the file input_x. If another node z is also an outgoing neighbor of y, then the controller will also append this very same message to the file input_z. In this way, even though y is not aware of who are its outgoing neighbors, the controller is aware of them, and will copy the message to the appropriate input files of the outgoing neighbors.

Note that this also will have that each file will be written by only one process (necessary due to the file locks used in Unix). E.g., output_x is written to by x and read by the controller. File input_x is written to by the controller and read by x.

Note that when x reads input_x there might not be anything else to read (you read an end of file), however, later the controller may write something new. Hence, the fact that you have reach an end of file that does not mean everything is over, the next time you read it the controller may have appended something new, and hence the next time you read you will get the new messages writen by the controller, and not an end of file.

The toplogy file will be simply called "topology" and the controller opens it at the beginning of its execution. The topology file consists simply of a list of pairs, each pair corresponds to a unidirectional channel. E.g, if we have three nodes, 0, 1, and 2, arranged in the form of a unidirectional ring, the topology file would contain

```
0 1
1 2
2 0
```

If the topology consists simply of two nodes 0 and 1, with bidirectional channels between them, the topology file would look like

```
0 1
1 0
```

Each receiver R will open a file called R_received_from_S where R is the receiver's ID (0.. 9), S is the ID of the sender whose tree R will join. Whenever R receives the string from S, it will write this string to this file. If R receives the string multiple times then it will write the string as many times as it receives it.

Hello Protocol

Every 5 seconds, each node will send out a hello message with the following format

hello ID

ID is the ID of the node sending the message (0 .. 9).

Note that if x receives hello messages from y, it knows there is a channel from y to x.

If after 30 seconds the node does not receive messages from some neighbor, it believes the node is no longer a neighbor (either it moved away or it is dead). This will have the obvious consequences for routing, i.e., any destination that was reachable via the neighbor is considered no longer reachable.

Routing Protocol

The routing protocol is a modification of distance-vector routing. DV routing does NOT work in unidirectional networks, so we have to modify it.

Periodically, every 5 seconds, each node sends out the following message

dvector sender-ID originator-ID d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 in-neighbors n1 ... nj where sender-ID is the id of the node that is sending the message, originator-ID is the ID of the node that originated this message (as we will see later this message will flood the entire network), d0 .. d9 are the distances (# of hops) from node originator-ID to each of nodes 0 ... 9, and n1 ... nj is a list of "in-neighbors" of node originator-ID. A node x is an "in-neighbors" of node y if there is a channel from x to y, i.e., if y receives hello messages from x.

If the node does not know how to reach another node then infinity will be used for the # of hops, you can use -1 to represent infinity.

Note that when the originator sends out the dvector message, then sender-ID = originator-ID

Consider neighbors x and y such that there is a channel x --> y, i.e. a channel from x to y. If y can reach some destination, then so can x by choosing y as its next hop. However, if the channel is unidirectional from x to y, then x will not receive a distance vector from y and thus x is unable to use y as a next hop to any destination.

To solve this, we would like the distance-vector of y to reach x. To do this, we do a **flood**. That is, the dvector message will reach x, and then x notices that it is in in-neighbor of y, and thus it can use y as a next-hop to reach some destinations. Thus, it can decide if going via y gives x the optimal path to the destination or not.

The way we will perform the broadcast is as follows. It will be similar to RPF, i.e., dvector messages that were sent by a node will only be forwarded if received from a specific neighbor.

In particular, a node x will forward the dvector message originating from node z only if the message was sent by neighbor y, if the number of hops from z to y is less than the number of hops from z to any other neighbor of x (ties are broken in favor of least node id).

Thus, if y is along the shortest path from z to x, then x will forward the dvector message that originated at z and was forwarded by y.

How will x know that of all its neighbors, y is the one along the shortest path from z to x?

Every node, every 5 seconds, will send out a message

in-distance ID d0 d1 d2 d3 d4 d5 d6 d7 d8 d9

where ID is the node sending the message (this message will NOT be flooded, it is just sent out from a node and all its neighbors receive it)

d0 .. d9 are what the node believes are the distances from nodes 0 ... 9 to itself.

By looking at the in-distance vectors of all its neighbors a node can determine (or update) what it thinks is the distance from other nodes to itself. Think of it as a distance-vector protocol in reverse, i.e., how others can reach ME, as opposed to how I can reach others.

Multicast Protocol

Tree construction

Assume a receiver R wants to join the multicast tree of a source S. R identifies the neighbor X whose distance from S to R is the least (this information follows from the "in-distance" messages received from its neighbors). Then, R sends a join message to neighbor X. Note that the link from X to R may be one-directional. Thus, the message may have to follow a different path to get to X.

The routing algorithm above is complete: if there is a path to X from R the routing algorithm will find it. Thus, R simply sends the message towards the neighbor which according to the routing tables is in the direction of X.

join ID SID PID NID

where ID is the ID of the node that wants to join the tree (i.e. R), SID is the ID of the root of the tree, PID is the ID of the parent on the tree (i.e. neighbor X), and NID is the next node along the path to reach PID.

When node NID receives this message, it will forward it to its neighbor NID' along the next hop to reach PID, thus it will forward the message as it is except it changes the last value

join ID SID PID NID'

A node chooses to join the tree if either it is a receiver, or if it is the parent of a node who also wants to join the tree.

When the parent receives this join message, it remembers who the child is, and then sends a join of its own to its parent on the tree (if you are not already on the tree)

A node that is on the tree will every 5 seconds send a join message to its parent.

If a node fails to receive join messages from its child for 30 seconds it assume the child is no longer on the tree.

Data Forwarding

Each source node S periodically (every 10 seconds) sends a message of the following form:

data sender-ID root-ID string

where sender-ID is the ID of the node sending the message, which in the first hop is S, but this value changes at every hop, and root-ID is the root of the tree, i.e., source S, and string is the string given in the arguments of the command line of S.

When a node receives a data message, it checks if it is coming from its parent along the tree rooted at root-ID. If the node is not a member of this tree, or if the message is not coming from its parent, the message is dropped. If the node is on the tree and the message comes from is parent, the message is forwarded.

Program Skeleton

The main program should look something like this

```
main(argc, argv) {

  Intialization steps (opening files, etc)

  for (i = 0; i < duration; i++) {

     send_hello();  /* send hello message, if it is time for another one */
     send dvector(); /* send a "dvector" message if it is time to do so */
     send in_distance(); /* send an "in-distance" message, if it is time to do so */
     refresh_parent(); /* send join message to each parent of each tree I am involved in, if time to do so */
     read_input_file (); /* read the input file and process each new message received */

     sleep(1); /* sleep for one second */
```

```
   }

}
```

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!

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 output files by hand (otherwise their contents would be used in the next run, which of course is incorrect). Note that at the end of each run, the files contain a trace of every message sent by each node.

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!