CS 6390 Spring 2013 Programming Project

NOTE: your project MUST run on one of the Unix machines on campus, NOT ON YOUR PC. I recommend {cs1,cs2}.utdallas.edu (linux machines) or {cs3,cs4}.utdallas.edu (solaris machines). Machines like apache will likely complain about resource usage!

You can use any language that you want, as long as it runs on the unix machines on campus.

I recoomend that you DO NOT use threads! In the past, students unknowingly have created around 200 threads which of course causes the system to kill your programs.

You will submit your source code and any instructions on how to compile it (i.e. which machine you compiled it on and using what command) You have to submite a README file, along with your source code, indicating: on which machine you run it, *exactly* what command you used to compile it.

Please make sure your program uses the arguments as described below and in the order described. Otherwise, the TA may have to modify the shell scripts to run our tests, which we will not be happy about (hence, possible point deduction).

We will run your code in the UTD unix machine that you mention in the README file, and see if it works.

This is a long description, so it can have many omissions, typos, and mistakes. The sooner you read it and find them, the sooner I will be able to fix them :)

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 link-state routing protocol, and then build a multicast protocol on top of it (and no it is not MOSPF)

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++, for example, or

node.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 &

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, and "&" indicates that the process will be run in the background (freeing up the terminal to do other things). E.g.,

> node 5 &

executes node 5 in the background.

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

> node ID sender string &

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"

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

> node ID receiver S &

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

> node 3 receiver 8 &

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 &

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 a configuration file (which you write before the execution of the simulation) that contains the network topology, i.e., the file describe 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.

A final note on input and output files, when a message is written to them, the message is APPENDED to the file. I.e., these files DO NOT shrink in size during the simulation. Actually, at the end of the simulatin the files will contain a record of every message sent by the node and every message received by the node. We will look at these files and they will be part of our grading process, i.e., make sure each node was sending and receiving the correct type of messages.

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).

Not that if x receives hello messages from y, it knows there is a channel from y to x.

Routing Protocol

The routing protocol is simply a link-state routing protocol (by the way, link-state works just fine in unidirectional networks). You should be familiar with link-state routing from your previous course.

Periodically, every 10 seconds, each node sends out a message (i.e. a linkstate adverstisement) of the following form

linkstate ID TS n-ID n-ID ...

where ID is the ID of the node that created this message, TS is a two-digit timestamp (incremented by one every time the node creates a new linkstate advertisement, only two digits, 00 ... 99, because the program will not run long enough to wrap around), and n-ID n-ID ... is a list of incoming neighbors of the node (learned from the hello messages).

Each node remembers the largest timestamp received from each other node in the system. When a linkstate advertisement is received, and it has a larger timestamp than the largest one received thus far, then the node will remember the information in this linkstate advertisement, and then forward the advertisement (without modifying it in any way) to its outgoing neighbors.

Although each node generates link-state advertisements periodically, whenever it receives an advertisement from another node, it will immediately forward the advertisements to its neighbors. Hence, an advertisement, once generated, will quickly propagate through the network.

Thus, eventually, each node has a complete view of the topology.

Link-state advertisements have a lifetime of 30 seconds. I.e., if a node x does not receive a link-state advertisement from some other node i in 30 or more seconds, x removes the link-state advertisement of i from its database.

Multicast Protocol

Tree construction

Assume a receiver R wants to join the multicast tree of a source S. Since R has complete view of the topology, it knows the shortest path from S to R. This is the path that should be on the tree.

Assume that the shortest path from S to R has node M as the node previous to R along this path. I.e., the path from S to R goes from S to M and the next hop is from M to R. R knows about M since it has the complete view of the topology.

R will then send a join message to M. Note that M is in incoming neighbor of R. However, there may not be a direct link from R to M. Then how can R send a message to M? The answer is SOURCE ROUTING.

Since R knows the entire topology, it can obtain the shortest path from R to M, and then include this path in the message from R to M.

Thus, R would send a message of the following form

join ID SID PID id0 id1 id2 ... idn

where ID is the ID of the node that wants to join the tree, SID is the ID of the root of the tree, PID is the ID of the parent on the tree, and id0 .. idn are the intermediate nodes from ID to PID (not including ID and PID).

For example, ID would be R, SID would be S, PID would be M, and id0 ... idn are the intermediate nodes along the path from R to M (not including R and M). The intermediate nodes will make sure this message is forwarded along the path and eventually delivered to M.

M receives this message and learns it has to join the tree rooted at S, so it continues the process, it computes the shortest path from S to M, and finds the node N on the last hop of this path (i.e. last hop is N --> M), and then sends a join message to N asking it to join the tree, etc.

Eventually the entire path from R to S has joined the tree. If a node is reached that is already on the tree

MORE DETAILS ON SOURCE ROUTING

Each node will remove one id when it receives the message, otherwise join messages could loop around

forever).

E.g, assume ID = 3 (who wants to join), PID = 0 (who is the parent), SID = 9 (the root) and the path from 3 to 0 is 3 5 4 0. Then 3 sends the message

join 3 (ID) 9 (SID) 0 (PID) 5 4 (intermediate nodes)

All neighbors will receive this message, and all (except 5) will drop it. Only 5 accepts it, and notices that 4 is the next hop, so it sends out the message

join 3 (ID) 9 (SID) 0 (PID) 4 (intermediate nodes is now just 4)

All neighbors of 5 will receive this message, and all (except 4) will drop it. Only 4 accepts it, and notices that there are no more intermediate nodes, and since 0 is the PID, 0 has to be a neighbor of 4 (it should be :)), so 4 sends out the message

join 3 (ID) 9 (SID) 0 (PID) (intermediate nodes is empty)

Since there are no more intermediate nodes, all nodes except the intended destination of this message (i.e. the parent, i.e. 0) will drop the message. Process 0 will keep this message and realize that it has a child 3 who wants to join the tree rooted at 9.

END OF DETAILS ON SOURCE ROUTING

A node that wishes to join the tree of S (either because it is a receiver or because it received a join message from a child) will send its first join message to its parent on this tree immediately after it becomes aware it wants to join the tree (ie. receivers will do so immediately, intermediate nodes immediately after receiving the join message from the child).

However, even though a node wants to join the tree (e.g. the receiver), there migh not yet be knowledge of a path from S to the node (link-state routing has not finished propagating yet). The node has to wait until it has knowledge of a path from S to itself before it can send a join message.

Afterwards, subsequent join messages to the its parent on the tree of S are sent periodically every 10 seconds to "refresh" the information at the parent. If a parent does not hear from a child after 20 seconds it removes the child from its list of children for the tree of S.

Furthermore, if the network topology changes, (new link-state advertisements are received or link-state advertisements expire), the node must recompute the path from S to itself. If it finds that it should have a different parent, it sends a join message to the new parent.

Note that a node may be involved in more than one 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 < 150; i++) {

     send_hello();  /* send hello message, if it is time for another one */
     send_link_state_advertisement(); /* send a link-state-advertisement if it is time for a new one*/
     refresh_parent(); /* send join message to each parent of each tree I am involved in, if it is 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!

Notice that your process will finish within 150 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 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

How to Run your Project

Consider the following scenario, it looks like this

```
controller &
node 0 sender "funny message" &
node 1 &
node 2 &
node 3 receiver 0 &
```

You obviously don't have time to type all of that at the Unix prompt, especially if you are a slow typist

So, create a new text file, for example, call it scenario.sh

Put the above lines in the scenario.sh file

Put the topology in a file called topology

topology file:

```
0 3
3 0
3 2
2 3
3 1
1 3
0 1
1 0
1 2
2 1
```

Then at the Unix prompt, type

sh scenario.sh

That will cause a "shell" program to read the commands from scenario.sh and execute them, so this will case the 3 nodes above to be run in the background

You can check that they are still running with the command ps -aef| grep youruserid

You can have a different scenario file for each of scenario

At the end of each scenario, and before you run the next one, don't forget to delete all channel files and all "nodeXreceived" files, since you have to start each scenario "afresh". You may have to change the topology file also if you have a new topology.

Good luck!