CS 4390 Spring 2014 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: see the attached file Unix.txt for a list of machines you can use.

Due: May 10th 5pm

WHAT TO SUBMIT

- **You have to submit all your source-code**
- **include a README file stating in which Unix machine you run your project, and the precise command (flags and all) that you used to compile your program.**

THERE WILL BE NO DEMOS FOR THE PROJECT

WE will compile your code and run it. If it gives sensible results you get a good grade. If not, well....

To emphasize, your project must run in the unix machines on campus. We do not care at all if it runs in your laptop. Your full grade depends on it running in the unix machines on campus.

Overview

Processes, files, 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, or LESS, not all nodes need to be present.

Each process (i.e. node) is going to be given the following arguments

1. id of this node (i.e., a number from 0 to 9)
2. the duration, in seconds, that the node should run before it terminates
3. the destination id of a process to which the transport protocol should send data
4. a string of arbitrary text which the transport layer will send to the destination
5. a list of id's of neighbors of the process

We will have a **single program** foo.c (or foo.java, or foo.cc 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 are the arguments to the program.

Note that each node is a separate process, NOT a thread. If you want to use threads within a node is up to you, **but each node is a separate unix process**.

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 foo be your program that represents a node. The arguments of the program are as follows

```
foo 3 100 5 "this is a message"  2 1  &
```

(or if you use java,

```
java foo 3 100 5 "this is a message" 2 1 &
```

)

The following would execute the program foo, and the first argument is the id of the node (3), the second is the number of seconds the process will run (100), followed by the destination for this node (5), then the message string "this is a message", and ending in a list of neighbors (i.e. 2 and 1 are neighbors of 3)

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, and node 1 to send a message "this is a message from 1" to node 2. Also, assume 0 and 1 are neighbors, and 1 and 2 are neighbors. Then I would execute the following commands at the Unix prompt > (your prompt may, of course, be different)

```
> foo 0 100 2 "this is a message from 0"  1 &

> foo 1 100  2  "this is a message from 1"  0 2 &

> foo 2 100  2 1 &
```

(or if you use java,

```
> java foo 0 100 2 "this is a message from 0"  1 &

> java foo 1 100  2  "this is a message from 1"  0 2 &

> java foo 2 100  2 1 &
```

)

This will run three copies of foo 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 transport level message to anyone, and the list of neighbors is just node 1

Note that if you have 8 nodes, it is rather hard to type all of the above (it will take a long time), so put the commands in a file, lets call it scenario1.sh, and then run it using a shell. E.g., create a text file called scenario1.h, and add the following lines to it

```
 foo 0 100 2 "this is a message from 0"  1 &
 foo 1 100  2  "this is a message from 1"  0 2 &
 foo 2 100  2 1 &
```

(or if you use java

```
 foo 0 100 2 "this is a message from 0"  1 &
 foo 1 100  2  "this is a message from 1"  0 2 &
 foo 2 100  2 1 &
```

)

So scenario1.h has three lines. Then, at the unix prompt > (or whatever is your unix prompt) type

```
> /usr/bin/sh scenaro1.sh
```

that will run the three commands above (spawn all three processes quickly in the background) instead of you typing them individually at the unix prompt.

The bash shell will read the file scenario1.sh, interpret its contents (i.e. spawn all the processes) and end.

Then if you type

```
> jobs
```

you will see a list of all your programs running in background mode, and there you will see the three nodes running

The channels will be modeled via text files. File name "from0to1" corresponds to the channel from node 0 to node 1. Therefore, node 0 opens this file for **appending** and node 1 opens this file for **reading**. File name "from1to0" corresponds to the channel from node 1 to node 0, and process 1 opens this file for appending and process 0 opens this file for reading.

Since you append messages to files, you **DO NOT OVERWRITE** the channel contents. At the end of the execution we will have that every channel file (e.g. from0to1) will contain all the messages that were sent (from node 0 to node 1)

One last thing. The channel files should be empty (or nonexistent) before the simulation begins, so, before you start a simulation, manually delete all fromxtoy files. You could also incorporate this into your scenario file as follows

```
/bin/rm from?to?
 foo 0 100 2 "this is a message from 0"  1 &
 foo 1 100  2  "this is a message from 1"  0 2 &
 foo 2 100  2 1 &
```

Program foo (which represents a node) will contain a transport layer, a network layer, and a data link layer.

Overview of each layer

The data link layer will read bytes from each of the input files, and separate the bytes into messages. Each message is then given to the network layer.

The network layer will determine if this node is the destination of the message. If it is, it gives the message to the transport layer. If it is not, it gives the message to the link layer to be forwarded to the channel towards the destination (the destination may be multiple hops away)

The network layer will also perform routing.

The transport layer will do two things:

a. Send the string given in the argument to the appropriate destination (by breaking it up and giving it to the network layer)
b. Receive messages from the network layer: only those messages that are, of course, addressed to this node
c. Output to a file called "nodeXreceived" where X is the node id (0 .. 9). The contents of this file should look like this for node 2 in the example above:

    From 0 received: this is a message from 0

    From 1 received: this is a message from 1

That is, it should contain one line for every message received, it should say from which node the message originated and what was the contents of the message.

Don't forget to also erase these files in between runs of your code. Therefore, incorporate the line

```
/bin/rm node?received
```

at the beginning of your scenario (i.e. shell) files.

Detail of Each Layer

Datalink layer

Since the channel is a file, you can read one byte at a time form the file, however, you need to determine when the frame (data link message) begins and when it ends.

Each message is a sequence of bytes. The first byte is the start-of-header byte, denoted by an upper-case s, i.e., `S'. Following this are two bytes indicating the length of the message (in ASCII), the maximum message length is 99 bytes. This lenght inclues all bytes in the message, from the 'S' all the way to the check byte at the trailer. Following that is the payload of the message (the actual data). Following that are two bytes that contain a checksum. The value of the checksum is from '00' to '99', and it is obtained by adding together the ascii codes of all the bytes in the message, starting with the 'S' modulo 100

For example, a data link message could look like

S08abc81

Note that the data part of the message could contain an S. If the receiver loses track of the beginning/end of messages, which, by the way, I can simulate by feeding incorrect file contents to one of your programs, then the receiver should be able to recover by looking for the next S and check if the checksum is correct.

The datalink layer has two subroutines (the subroutines don't have to have exactly the same parameters, you can add more or change them if you want, they are just a guideline...)

a. datalink_receive_from_network(char * msg, int len, char next_hop)
   This function will be **called by the network layer** to give a network layer message to the datalink layer. The network layer message is pointed to by char * msg, and the length of the message is integer len. The argument next_hop is the id of the neighboring node that should receive this message. This routine will output to the output channel (text file) the message given by the network layer, with the format described above.
b. datalink_receive_from_channel()
   This function **reads from each of the input files** (i.e. the channels from each neighbor) until it reaches an end of file in each of these files. Whenever it has a complete message it gives it to the network layer by calling network_receive_from_datalink() (see network layer below). Again, the fact that it reached an end-of-file **does not** mean that there are no more bytes, since these bytes may not have arrived yet. Thus, you have to read until you get end-of-file, and then you have to put your program to "sleep" for 1 second. When it wakes up you should try to read more. If there is nothing to read you go to sleep for one second more, etc.. (See Program Skeleton below) Also, note that the fact that one file has no more data does not mean that there is no more data from other files. Thus, once you reach an end-of-file on ALL input files, then you go to sleep for a second

Network Layer

The network layer will find routes between nodes and forward data messages along their path to the destination. It will have two types of messages, data messages (which carry a transport layer message inside) and configuration messages (which help compute routes).

Data messages have the following format

a. 1 byte for message type: "D" for data
b. 1 byte source id (from "0" up to "9")
c. 1 byte destination id (from "0" up to "9")
d. up to 10 bytes; these bytes are a transport layer message.

Data messages are simply routed towards their destination. The trick is to find the path to the destination.

By the way, if a message is addressed to a neighboring node, you send the message directly to your neighbor, regardless of the routing tables say. Hence, two neighboring nodes should be able to communicate even if the routing protocol does not work well.

We will perform routing in a very similar (if not identical) way in which Ethernet bridges perform routing. Thus, we first have to build a spanning tree of the network. Once we have the spanning tree, we route messages by flooding them in all directions along the tree (ONLY ALONG TREE EDGES).

We will not learn the way bridges do. We could, but it will make things more difficult for us (trust me, especially if the topology changes by killing a node, which will be one of my test scenarios). So, we simply flood messages in all directions.

We build the spanning tree by sending configuration messages of the form:

a. 1 byte for message type: "C" for configuration message
b. 1 byte for the root ID ("0" up to "9")
c. 2 bytes for the hop count (from "00" up to "99"). The hop count will likely not go beyond 9 or 10 (depending how to code it) but let us use 2 bytes anyway.

Initially, every node thinks it is the root, so it sends configuration messages to each of its neighbors.

The root (or whoever thinks is the root) sends configuration messages to all its neighbors evey 5 seconds.

If a node does not think of itself as the root, and if no configuration messages have been received in 20 seconds with a root ID smaller than itself, then the node chooses to become the root.

The algorithm of course is greedy, trying to find a parent that gives you the smallest (root ID, hop count, neighbor id) triplet of values. I.e., the root ID is the most important value, then the hop count, then the neighbor from whom you received the information. Note that this ensures that for a given topology there is one and only one possible spanning tree.

When a node changes parents (or becomes root) it assumes all its neighbors (other than its parent) are its children. If the node then receives a configuration message from a child and the message has the same root ID and the same hop count, then this node is removed from the list of children (because its information is as good as mine and hence it is using some other node as its parent and not me).

The network layer has three subroutines

a. network_receive_from_transport(char * msg, int len, int dest). This function is called by the transport layer. It asks the network layer to send the byte sequence msg of length len bytes as a message to destination dest.
b. network_receive_from_datalink(char * msg, int neighbor_id) this function is called by the data link layer, indicating that a message msg was received from the channel from neighbor neighbor_id. If the message is a data message, and it is addressed to this node, then the network layer gives the message to the transport layer by calling transport_receive_from_network() (see transport layer routines below). If it is addressed to another node it floods it to all other neighbors (not where it came from!). If it is a configuration message, it processes it as discussed above.
c. network_check_if_root() - this is called by the main program (see below). If the node does not considers itself the root, then it checks if 20 seconds have ellapsed from the last time the node received a configuration message. If no configuration messages have been received in 20 seconds, the node chooses to become the root. If the node is the root, it will send a configuration message to each of its neighbors every 5 seconds.

Transport Layer

Each transport layer message will be limited in size, and hence, if the string given as argument to the program is bigger than this, then the string will have to **be split into multiple transport layer messages**.

The transport layer will implement a negative acknowledgement protocol. (No we did not cover it in class, but it is a simple one).

Data messages have the following format

a. 1 byte for message type: "D" for data
b. 1 byte source id (from "0" up to "9")
c. 1 byte destination id (from "0" up to "9")
d. 2 byte sequence number (from "00" to "99")
e. up to five bytes of data (i.e. up to five bytes of the string to be transported)

Negative acks messages have the following format

a. 1 byte message type: "N" for nack
b. 1 byte source id (from "0" up to "9") (the receiver)
c. 1 byte destination id (from "0" up to "9") (the sender)
d. 2 byte sequence number (it contains the sequence number of the message that was not received).

Source Behavior:

The source of the data will try to trasfer the "string" that was given as argument to the program to the destination which was also given as argument to the program

The string will likely be more than five bytes, so break it up into five-byte pieces, each becomes a data message and has a sequence number.

ALL of these data messages are sent to the destination (via the network layer).

If the source receives a nack for a particular data message, it retransmits this particular message.

If the source receives a nack with sequence number larger than the largest sequence number of the data (call this the "final" nack), then it is done, it assumes all data was received.

If the source does not receive a nack for a period of 20 seconds, and we have not received the final nack, it resends all of the data messages again.

Destination Behavior

The destination does not know who it will receive data messages from until the messages begin to arrive.

Data messages will have sequence numbers 00, 01, 02, 03, etc.

Assume you receive data message with seq # i at time t. If by time t+5 (5 more seconds) you have not received all data messages in the range 0 to i, then you send a nack for every data message that your are missing in this range.

Also, assume you receive data message with seq# i at time t. If by time t+5 you have not received data message with seq# i+1, then you send a nack i+1 back to the source.

The transport protocol has two subroutines: (the subroutines don't have to have exactly the same parameters, you can add more or change them if you want, they are just a guideline)

a. transport_send().
   This routine is called once per second **by the main program**. The transport layer sends the data to

the destination. Also, if this node is a receiver (it has received data) then it also sends nacks as appropriate.

b. transport_receive_from_network(char *msg, int len, int source)
Receives a transport-layer message from the network layer. This function is called by the network layer when it has a message ready for the transport layer. The transport layer message is pointed to by msg, and the messge length is len. The argument source indicates the original source of this message (i.e. the source field in the network layer message). .

c. transport_output_all_received()
This function is called only once at the end of the program. It will **reassemble** the string that it received from each source, then output the string into the nodeXreceived file, where X is the id of the node (as described ealier).

Program Skeleton

The main program simply consists of two subroutine calls:

1. A call to the transport layer to send messages
2. A call to the data-link layer to receive messages from the input channels

It should look something like this (well, in general, and depends on the language you choose, of course)

```
main(argc, argv) {

Intialization steps (opening files, etc)

let life = # seconds of life of the process according to the arguments

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

  datalink_receive_from_channel();

  transport_send();

  sleep(1);

}

transport_output_all_received()

}
```

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!

<u>Notice that your process will finish within "life" seconds (or about) after you started it.</u>

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).

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
```

In the assignment I have also included a little writeup on Unix (how to compile, etc). However, you should have enough info by now to start working on the design of the code. I have also included in the assignment some sample scenarions for you to run your program and get an idea of what grade to expect.

Good luck!