CS 4390 Spring 2023 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 to do your PROGRAM DEVELOPMENT, and to RUN the project I recommend net01, net02, etc (I think there are like 20 different netXX machines). The netXX machines have settings more suitable for the project (in terms of resources allowed per user). Other machines 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

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. the starting time for the transport layer (explained much later below)
6. a list of id's of neighbors of the process

We will have a **single program** node.c (or node.java, or node.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.

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

node 3 100 5 "this is a message" 40 2 1

The following would execute the program node, 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", then the time at which the transport layer begins, and the last arguments is 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)

> node 0 100 2 "this is a message from 0" 30 1 &

> node 1 100 2 "this is a message from 1" 30 0 2 &

> node 2 100 2 1 &

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 transport level message to anyone, and the list of neighbors is just node 1. Note that it does not have a start time for the trasnport layer

Note that each node can be the source of a string or not be a source at all. However, each node may receive strings from many other nodes. You cannot know from whom in advance.

The channels will be modeled via TEXT files (when we grade, we should be able to read them via a typical text editor, the files should contain only normal characters, i.e., digits, letters, spaces, etc.). File name "from0to1.txt" corresponds to the channel from node 0 to node 1. Therefore, node 0 opens this file for writing (actually, **appending**, you don't overwrite any previous contents) and node 1 opens this file for reading. File name "from1to0.txt" corresponds to the channel from node 1 to node 0, and process 1 opens this file for writing (appending) and process 0 opens this file for reading.

Note that at the end of the execution, when all nodes have finished, the file "from1to0.txt" will contain all the messages that were sent from node 1 to node 0 during the execution. I.e., messages are **appended** to this file, they are never deleted. We will look at your channel files at the end of the execution to check that all the messages were sent correctly.

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

Overview of each layer

The data link layer will read bytes from each of the input files. The input files are just a sequence of bytes. The datalink layer has to 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 the following:

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) all the stings that you received. 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

The datalink layer will do the following:

1. 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 (message) begins and when it ends. So, we will use fixed-sized frames with a start of message sequence and a checksum to check for correctness.
2. When a whole message is received, give it to the network layer.
3. Receive messages from the network layer, encapsulate them into data link layer messages, and append them to the appropriate text file.

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 (message) begins and when it ends.

Each message will begin with the bytes "XX", followed by 15 bytes of data, followed by two bytes for the checksum

Each network layer message carried inside the datalink layer message will be of fixed size (15 bytes).

The checksum will be a two byte integer (in ASCII, hence two bytes, NOT in binary), i.e. from 00 to 99. This number will be the sum of the 15 bytes of the data, modulo 100 of course.

If a message is corrupted, (checksum is incorrect), then the datalink layer scans for the next XX (hopefully is at the next two bytes, but it may not if some bytes were accidentally deleted), and attemps to find the correct location for the beginning of the next message. **Note that the data itself could contain the XX value,** and hence, the next XX value you see may or may not be the beginning of the next message. If XX occurs 19 bytes and the checksums appear ok then you can assume that you have "resync'ed", i.e., you have found the right place in the byte stream where each message begins.

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, 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 fixed (as

described above). The 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, as 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 handle two types of messages, "data" messages and "routing" messages. Data messages are used to carry transport layer messages. Routing messages are used to find the routing path to each destination (to each of the other 9 nodes).

Data messages consist of the letter "D" followed by 1 byte for the destination, two bytes that count the number of data bytes in the data message, followed by the data itself, which are the contents of the transport layer message being carried by the network layer.

Note that the transport layer message may not be big enough to make the whole data message 15 bytes long. Hence, we need to pad it before giving the message to the datalink layer. You can pad the message with spaces (blanks, " ").

For the routing protocol, we will implement a link-state routing approach. Each node, every 10 seconds, will generate a link-state packet (LSP), and give a copy of it (via the datalink layer of course) to each of its neighbors.

The LSP has the following format.

- The letter "L" (for link-state packet). This distinguishes it from a data packet, which begins with D.
- The Id of the node (one byte, "0" through "9").
- A two digit sequence number (or timestamp), "00" thru "99"
- A list of its "live" neighbors
- Padding as necessary (spaces).

Every 10 seconds, when the node generates a new LSP, it increases its sequence number. Hence, the first LSP has a seq # of 00, the next one generated 10 seconds later is 01, etc

All neighbors are assumed to be live at the beginning of the program. If a neighbor has not sent a message to this node (of any type, data or link-state packet) then we declare the neighbor dead and remove it from our live list

The neighbor can be added to the live list again if the node receives again a message from the neighbor.

As in a typical routing protocol, the node should construct a topology graph, and find the shortest path (minimum hops) to all other nodes.

To make routing consistent in the project, if two neighbors are the same # hops to a destination, choose as next hop the one with the lowest id.

Note also that even if your routing protocol is not working properly, you should always be able to send data messages to your neighbors, since you know them in advance from the arguments of the program.

The network layer has two 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 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)
c. network_route() - this is called by the main program (see below), and it checks if 10 seconds have elapsed from the last time the node send its routing messages. If so, it sends a new LSP out to each of its neighbors.

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. If enough time has ellapsed (see arguments of 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 thenodeXreceived 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!

Notice that your process will finish within "life" 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).

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 program

You will notice that to run a scenario in your network, there is a lot of typing! For example, the small scenario above:

```
> node 0 100 2 "this is a message from 0"  30 1 &

> node 1 100  2  "this is a message from 1"  30 0 2 &

> node 2  100  2 1 &
```

requires you to type a lot! It is worse of course if we have 10 nodes.

What we want is to start all nodes at the same time (or very close to the same time). So, we use the shell program to do this for us. To run a scenario, create a file, e.g., scenario.sh, and add the following to it with a text editor

```
node 0 100 2 "this is a message from 0"  30 1 &

node 1 100  2  "this is a message from 1"  30 0 2 &

node 2  100  2 1 &
```

Then, execute the following at the unix prompt

```
>  sh scenario.sh
```

This should put all three processes in the background. You can double check this by doing the command "jobs"

```
> jobs
```

You should see that all of the three nodes are running in the background.

That's it.

Good luck !!!