CS 5390 Spring 2019 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.
You can use any language that you want, as long as it runs on the unix machines on campus.

YOU CANNOT USE THREADS. People using threads (especially in Java) have encountered problems in the past, so stay away from threads please.

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** 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.

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"  40 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", 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)

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

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

```
> 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 it does not have a start time for the trasnport layer

The channels will be modeled via TEXT files (when we grade, we should be able to read them via a regular text editor). File name "from0to1" 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" 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.

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

The datalink layer will do two things:

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 byte-stuffing (a.k.a byte insertion) for this.
2. In addition to marking the beginning and end of a frame, the datalink-layer makes sure that messages are sent reliably to the next-hop. It will use the concurrent logical channels protocol to do this.

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.

We will use byte-stuffing (a.k.a byte insertion) for this.

The beginning of a frame will be indicated by the letter F (for frame :) the end of the frame will be denoted by the letter E. If an F or an E appear anywhere in the frame, they have to be escaped with the letter X. If X occurs in the frame then of course it is also escaped with another X.

In addition to marking the beginning and end of a frame, the datalink-layer makes sure that messages are sent reliably to the next-hop. It will use the concurrent logical channels protocol to do this. Thus, there are two type of datalink frames: data messages and ack messages. Data messages carry inside of them a network layer message. Acks, well, are simply acks. The datalink-layer will use two logical channels, channel 0 and channel 1.

data frames have the following format:

data x y <netw layer message>

where x is the channel number, i.e. 0 or 1, and y is the sequence number of the message being acked and <netw layer message> is a message from the network layer.

ack frames have the following format:

ack x y

x and y are the same as above.

A timeout value is needed when waiting for an ack. This value will be 5 seconds.

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 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, and they are of varying length. Routing messages are sent between neighbors to find a path to each destination and are of fixed length: 12 bytes.

Data messages consist of the letter "D" followed by 1 byte for the destination, and then the bytes for the transport layer message (that are carried inside the network layer message)..

The routing protocol will be simple. Each node will try to find the shortest path to each destination. Consider e.g. node 5. Periodically, namely, every 5 seconds, node 5 will send 10 routing messages to each of its neighbors. Each routing message indicates the path from node 5 to the destination. Routing messages begin with an "R" followed by a path that ends in the destination, and ending in an X.

E.g., the message

R 5 2 6 9 X X X X X X X

Indicates that the path to reach 9 is 5 2 6 9 and the message

R 5 1 3 6 7 9 0 X X X X

indicates tha the path from 5 to 0 is 5 1 3 6 7 9 0

If node 5 is not aware of any path to node 8, it will indicate it with a message

R 5 X X X X X X X X X 8

Note that this is different than the message that says that 5 and 8 are neighbors, which would be like

R 5 8 X X X X X X X X X

Obviously, nodes should behave in a greedy way, when you find a path via some neighbor that is better than the current path you have chosen, then you change your path to the shorter path.

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 len, 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 5 seconds have elapsed from the last time the node send its routing messages. If so, it sends the 10 routing messages 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 simple forward error recovery protocol, i.e., we only have data messages not ack's. For every pair of data messages sent, we will also send the exclusive or XOR of the previous two data messages. Thus, if we lose one of the data messages, you can recover its contents from the other message in the pair and from the XOR message.

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)

XOR messages have the following format

a. 1 byte message type, whose value is "X" for XOR
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 (it contains the sequence number of the even message in the pair)
e. up to five numbers (separated by a blank) in the range 0 .. 255, these correspond to the XOR of the corresponding data bytes

For every pair of (even, odd) data messages, you will send the XOR of the bits in the message. Note that the XOR would yield some strange binary numbers, and since we want to be able to see the contents of your channels (text files) with a regular text editor, then you cwill write a number 0 .. 255 for every byte.

NOTE:

- The transport layer cannot send its messages until enough time has ellapsed, which is indicated in the arguments of the program.
- Each node may receive strings from multiple senders at the same time. E.g., nodes 0 and 1 may be at the same time sending a message to node 2, so node 2 is acting as a receiver for two senders. Notice also that node 2 may at the same time have a string that it is trying to send to node 0 (or another node), so a node may act, at the same time, as a receiver that receives from multiple nodes and as a sender that sends to one node.

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_string().
   This routine is called at the appropriate time (after enough time ellapses). It takes the string given in the argument of the program, cuts it into appropriate sized pieces, and sends it (and the XOR messages) to the destination (via the network layer of course).
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). The message received could either be a data message or an XOR, and thus mut be treated accordingly.
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 the data messages (after enough time ellapses)
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)

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();

```
   if (i > time to begin transport) then  transport_send_string();

   sleep(1);

}

}
```

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
```

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