Machine Problem 2

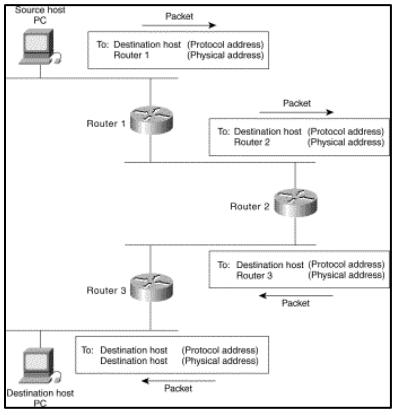
Unicast Routing

Due: Mar 15th, 2012 – 11:59 PM

Please read all sections of this document before you begin coding. There are many hints and suggestions that can save you time if you read them before designing the structure of your code.

One of the key problems in a packet switched network is routing: the process of deciding where to forward packets to make them reach their destination. In this machine problem, you will implement a distributed, dynamic unicast routing protocol. It is distributed because your protocol will run as separate processes each simulating a router, discovering the network topology and disseminating that information throughout the network, no central component has global knowledge of the network and all decisions are local. It is dynamic because your protocol will have to deal with a changing network, including link failures and link cost changes while continuing to successfully deliver as many packets as possible. It is unicast because it deals with the common case, sending each given packet from a single source to a single destination.

In this MP you will write a program that simulates a simple router. Routers are generally physical devices, connected using cables that forward packets from a source to a destination through multi-hop links:



They generally measure a "cost" for each of their links, which can be Round Trip Time, Energy, or other metrics. In a large network, many paths traversing different subsets of routers connect a source and a destination node. The goal of a routing protocol is to deliver packets choosing one of these paths, generally trying to minimize the total cost of it. Large networks are also dynamic: the link cost changes with time and congestions, and hardware failures often make links unavailable. We will provide a manager program that connects to each simulated router and coordinates the simulation, describing the network topology and its changes, generating traffic and monitoring the path that messages take on their way to the destination.

This document specifies how your router should interact with our manager process to send and receive packets and be informed of the state of links. But the routing protocol is entirely up to you: you can use one of several classics discussed in lecture, or even invent one of your own.
The goals of this MP are for you to:

- gain direct experience with the design and implementation of a routing protocol, and the challenges in dealing with dynamics in a widely distributed protocol;
- improve your experience with threads and socket programming;
- learn how to use the Linux networking tools to debug your projects;
- learn to collaborate with other programmers in writing code, starting from the design phase.

You have more than a month to complete this Machine Problem. If you pace yourself well from the beginning, you will have no problem finishing it on time.

This MP is about routing protocols, which we have not already presented in class. However, the first steps do not require any routing strategies. If you start early and follow the assignment, you will get to the routing part at the right moment, and you will have plenty of time to complete the hardest part of the MP.

We give you complete freedom in how you design your code, but this does not mean that we are not willing to advise. If you start early and pace yourself properly, you can come and talk to us during office hours in the first weeks. We offer help not only to solve your coding issues, but also for your design choices.

Specification

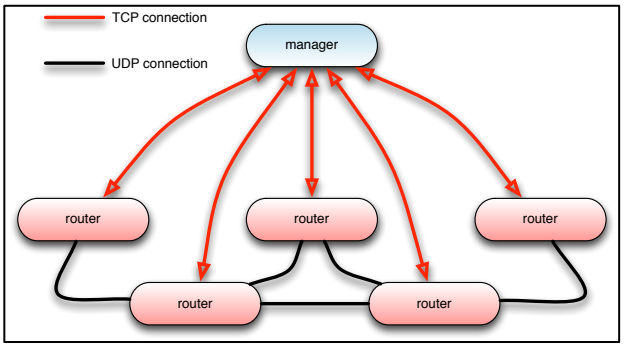
As always, we recommend that you use a 64-bit EWS machine since our auto-grader program will check your MP submissions on these machines**.** Information about the workstations is available on the homepage:

`http://ews.uiuc.edu/`

Update your SVN directory using `svn update` (if you are not familiar with SVN check the instructions on Piazza). You will find a folder named `_projects/<groupname>/MP2`, which contains a java archive (.jar). This file is the core of our simulation; you will have to write a program in C, which interacts with our manager. Your code simulates one router. We will test your code on several network topologies. For each topology, we will launch one instance of your code per each node in the simulated topology. Each node must have two connections: one TCP socket connects to our manager and receives commands during the simulation; one UDP socket is instead used for message passing among the nodes of our "simulated" network.
For this MP, you are encouraged to work in teams of two, although you may work alone if desired. We recommend that you collaborate on the infrastructure, while each partner **MUST** choose a different protocol to implement. Please find a partner as soon as possible, if you haven't done so or decided to work alone, already. If you make your lack of a partner known to us, we will pair you with another person or will make suitable accommodations. If you do not, you may have to work alone. In any case, we will release your MP2 folders only after you register your "group", be it a partnership or an individual effort. Instructions on how to register your group are found on Piazza.
Collaboration, discussion, code sharing, and any other form of group work with persons other than your MP2 partner is forbidden. Please indent and document your code. Use meaningful names for variables and follow other style guidelines that enhance the clarity of your code. Follow the guidelines in the HandIn section below when turning this assignment.

Project Description

You will simulate a network topology where each node is a process. Your program should simulate a single node; you will run an instance of your router per each node in the network. Our manager program has a TCP connection with each router instance. This is the control channel. Each router also implements a UDP server that emulates the physical connection between routers. The data exchanged among routers will travel through these UDP connections only, none of the messages required for your routing protocol can pass on the TCP connections. The following is a schematic representation of the entire simulation architecture:



The execution of the simulation begins by starting our manager:

```
java –jar manager.jar –p <tcpport> -t <topologyXMLfile>
```

The manager takes as command line arguments the TCP port on which it will be listening (`-p`), and the path to an XML file that describes the simulation (`-t`). We provide a few examples of XML files with simple topologies that you can use for your initial test. We might provide more complex topologies later, and of course we suggest that you create your own test cases. Once the server is running, you must launch a number of routers equal to the number of nodes simulated in the current topology (more on the syntax of the XML file later).
When you begin, we suggest that you run a simulation with a limited number of nodes (at least 3), and that you run each node in a separate console. This will help you to debug your code in the initial phase. We will see later how to test your code with a large number of nodes. Your router must require three command line arguments:

```
./router hostname managerport listenon
```

- `hostname`*:* is the hostname on which the manager is running. Throughout this assignment, we will always run it on localhost, but if you do things properly, the same code should work also if run across different machines.
- `managerport`: is the TCP port on which the manager is listening to
- `listenon`: is the UDP port on which the router should listen for incoming messages from other routers.

The simulation starts with an initial phase during which all the router instances establish a TCP connection to the manager. This TCP connection has three goals:

- during the initial handshake, the manager provides each node with a local description of the network (i.e., the nodes that are directly connected to it and the cost for each link);
- once all the nodes have completed the handshake, the simulation begins, and the manager uses the TCP channel to inform the routers about changes in the topology, which must be promptly accepted;
- during the simulation, you are required to use the TCP connections to send log messages to our manager that will be used to grade your algorithm.

Your router should also implement a UDP server, listening on the port specified with the command line argument. You can choose any port number; if you work on a shared machine like in the EWS lab you might pick one which is in use already by another user, simply restart your router with a new port. Selecting random numbers should make this occurrence quite rare. The UDP socket simulates the actual network interface of a real router. Through the UDP port messages will be forwarded from a source node to a destination node, passing through intermediate routers if necessary. We learned how routing protocol work in class, now let's see how we are going to simulate one in detail.

The XML file

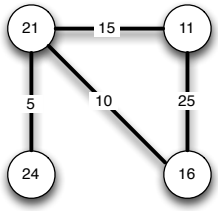
First of all, let's see the XML syntax for the file that describes the network topology. The following is a simple example that contains all the necessary elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulation>
	<Topology>
		<Link>
			<node>11</node>
			<node>21</node>
			<cost>15</cost>
		</Link>
		<Link>
			<node>21</node>
			<node>24</node>
			<cost>5</cost>
		</Link>
		<Link>
			<node>16</node>
			<node>11</node>
			<cost>25</cost>
		</Link>
		<Link>
			<node>21</node>
			<node>16</node>
			<cost>10</cost>
		</Link>

	</Topology>
	<Events>
		<Message>
			<src>11</src>
			<dst>24</dst>
			<data>asdf</data>
		</Message>
		<Message>
			<src>24</src>
			<dst>16</dst>
			<data>qwerty</data>
		</Message>
		<Link>
			<node>11</node>
			<node>16</node>
			<cost>-1</cost>
		</Link>
		<Link>
			<node>21</node>
			<node>11</node>
			<cost>5</cost>
		</Link>
		<Message>
			<src>16</src>
			<dst>11</dst>
			<data>fghj</data>
		</Message>
	</Events>
</Simulation>
```

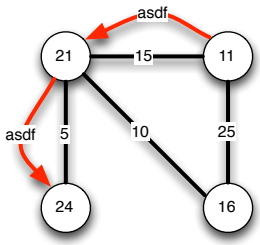
The root node of the file is the `<Simulation>` element. At the second level there are the elements `<Topology>` and `<Events>`. The first section describes the initial topology. Each `<Link>` node entry defines a possible communication channel between two routers (the `<node>` item) and a cost for that link (the `<Link>` item). A negative cost means that the link is temporarily unavailable.

The second section is the `<Events>` section. This section lists the messages that should be sent, and the changes in link cost during the simulation. A change in link cost is described with the same `<Link>` item described above. All links must be listed in the topology section, if they exists but are temporarily disabled, they will be listed with a cost equal to -1. A message transmission is listed in the `<Message>` item. Each message goes from `<src>` to `<dst>` and carries the payload defined in the `<data>` element (we use ASCII values for simplifying the debug). Events are simulated in a sequence, each event happens after the previous one has completed. A change in link cost is completed when both nodes have been informed and have acknowledged the new cost. A message transmission is completed when the intended destination reports that it has received the packet. The following is a representation of the network topology as defined by the Topology section:



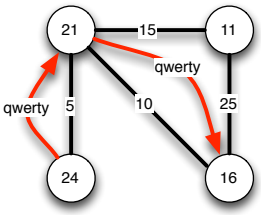

```
<Link>
        <node>11</node>
        <node>21</node>
        <cost>15</cost>
</Link>
<Link>
        <node>21</node>
        <node>24</node>
        <cost>5</cost>
</Link>
<Link>
        <node>16</node>
        <node>11</node>
        <cost>25</cost>
</Link>
<Link>
        <node>21</node>
        <node>16</node>
        <cost>10</cost>
</Link>
```

The numbers on top of each link represent the cost for that specific link. Now let's see what happens when we start the simulation. At first, a message is sent from 11 to 24:



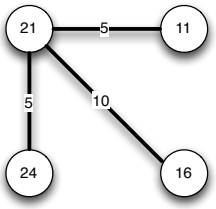

```
<Message>
        <src>11</src>
        <dst>24</dst>
        <data>asdf</data>
</Message>
```

followed by one sent from 24 to 16:



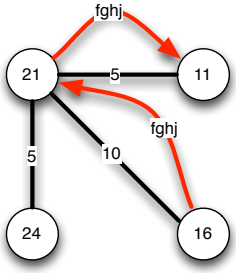

```
<Message>
        <src>24</src>
        <dst>16</dst>
        <data>qwerty</data>
</Message>
```

Then the link configuration changes:




```
<Link>
        <node>11</node>
        <node>16</node>
        <cost>-1</cost>
</Link>
<Link>
        <node>21</node>
        <node>11</node>
        <cost>5</cost>
</Link>
```

Finally one last message is sent with this new configuration. Note that it would be wrong to send the message from 16 to 11 directly, because the old link is not working in the current topology.




```
<Message>
        <src>16</src>
        <dst>11</dst>
        <data>fghj</data>
</Message>
```

Of course what is displayed in the examples above, is just one possible route through which the messages can be forwarded.

Router

Ok, now that we understand the theory, it is time to start the actual implementation.
We will describe step by step how your router should work.

[Step 1 – The TCP connection]

The first thing you should do is starting the manager we provide. We select a TCP port for the manager, 3690 in this example, and run:

```
java –jar mp2manager.jar –p 3690 ./Topology_Only.xml
```

The server should print:

```
Event section missing in the XML file
Waiting for a node connection...
```

At this point the manager is up and listening for incoming connections from the clients. The topology file we are using describes a triangular topology, with each node having a direct link to the other two. We are going to test the initial phase of the handshake with a telnet client first, to see how it works. Open three terminals, and start a telnet session with the server on each of them:

```
telnet localhost 3690
```

(of course you must remember to use the port number the manager is listening on). Telnet opens a TCP connection to the manager. We designed the handshake protocol using only printable chars, so that before implementing it in your code, you can test it by manually typing the commands.
You must initiate the handshake. Simply type:

```
HELO\n
```

(must be uppercase and there must be no space before the newline char, networking protocols are strict!). The server replies with the following message:

```
ADDR addr
```

Where `addr` is the ID of one of the nodes described in the topology file. **This will be the address of your router from now on.** If you type the command in the three telnet terminals you have open, you will receive a message in each terminal with three different addresses. Now is time to get your hands dirty.
First kill the manager. This will also close the three telnet sessions. Then reproduce what you have just seen happening:

- Create the file `router.c`, which takes exactly three parameters: the address of the manager, the TCP port it is listening to, and the UDP port on which the router will be listening.
- Create a TCP socket and connect to the specified host and port.
- Once the connection is established, send the `HELO` message.
- Receive the manager response
- Print the following line on standard output:

```
manager replied with address addr
```

(replace *addr* with the address received from the server). Do not just print the address, store it in your program. You will need it later. Now compile and run in the three terminals. Pick three random UDP ports; in this example we will use 5555. Now in the three terminal windows run the router with the command:

```
./router localhost 3690 5555
```

If your code runs fine, and prints the three lines with the addresses that the manager assigned to each node, you have successfully completed [step1]. If you send a wrong message instead, the server will reply with an error message. NOTE: in the past we had reports of the keyword "localhost" not working on the ews machines. If this happens, you can use the ip address "127.0.0.1").

[Step 2 – The Handshake]

Now that we have seen the first message going through, let's see the whole handshake procedure. Kill and start the server again, and open the three telnet connections once more. We know already the first step of the handshake (m: is the manager, r: is the client, do not send these chars, nor the space after the colon):

```
r: HELO
m: ADDR addr
```

Save the address in a variable, and from now on every time you have to identify your router use this value.
The protocol continues with the following message:

```
r: HOST localhost udpport
```

Where `udpport` is the UDP port that we decided to use for this specific node. Since we are running everything in the local machine, it is safe for this MP to simply use the hardwired value `localhost` to specify which host the nodes should connect to. But if you do things properly, you can replace localhost with the real hostname, and you should have no problem running the simulation across multiple machines with no changes in your code. However, we will not test this scenario.
The manager replies to your HOST message with an acknowledgment:

```
m: OK
```

Now the router is ready to learn about its neighbors, and the cost of each of the links:

```
r: NEIGH?
```

Before the neighborhood list can be generated, the node must know on which port each client is listening. Thus, the answer to this request will come later, only when all the nodes have successfully completed the initial part of the handshake. The TCP connection will be unresponsive while the manager waits the HELO and HOST commands from all the nodes. Once all your routers have registered, the manager knows on which UDP port and hostname each router is listening, and can generate the topology description, which is sent to the node that sent the NEIGH? message:

```
m: NEIGH addr localhost udpport cost
m: NEIGH addr localhost udpport cost
m: DONE
```

where each line identifies clearly one of the neighbors, and the cost of its link. Each node receives ONLY the list of one-hop neighbors as described in the topology file. We will see later that it is your duty to distribute across the network the IDs of all the nodes and construct the multi-hop routes to reach them. For now, you must simply store these addresses in a data structure that is needed later. The node address can range from 0 to 65535, and they are not necessarily in a sequence (i.e., a simulation could have nodes 1, 54, 2345). Try to be smart with your memory, creating a 65k elements array would use too much memory, and in routers memory is a very scarce resource. A dynamic structure is preferable. When all the nodes have been listed, the manager informs the router with the message DONE. It then waits for a final confirmation from the router than everything is ready and that the simulation can run. This happens with the following message exchange:

```
r: READY
s: OK
s: END
```

Now go ahead and implement this too in your `router.c` file, then run it as described before. If you do something wrong, the server will reply with an error message. **If you receive the strings `OK` and `END` from the server and no error messages, you just completed the second step**.

[Step 3 – The Link Cost Events]

Now that we have completed the handshake, it's time to spice things up. We need to run again the manager, this time with a different topology file. Use the file `Topology_and_cost.xml`. Again, before implementing anything, let's see how it works with telnet. Perform the handshake on each node again. Now, when the last of your routers completes the handshake, the manager will initiate delivering the control messages for the rest of the simulation. In the XML file we are using there is only two events: both are changes of link costs. The manager will send the first one to the relative nodes, and once this is acknowledged, it will send the second one. The two ends of the link that is changing received the following message:

`m:` `LINKCOST` *`node1 node2 cost`*

where *`node1`* and *`node2`* are the two ends of the link, and *`cost`* is the new cost. Remember that a negative cost means that the link is down, and no data can be forwarded directly between the two nodes. But we will get to that step. To acknowledge the message, the router responds with this message.

`r:` `COST` *`cost`* `OK`

once both nodes acknowledge the first link cost, the manager sends the next one. These two events are all we have for now. The line:

`m:` `END`

informs all the router nodes that the simulation is complete. The routers can close the connection sending:

`r:` `BYE`

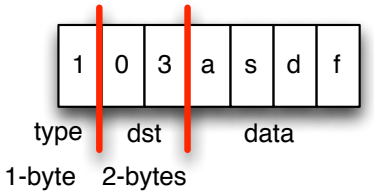
Of course your code must not simply reply to the cost information, but also update the cost value for the specified link in the data structure that was saved in step 2. You will need these cost to determine which route should be used to reach a node that is not a direct neighbor.
You should have no problems in implementing what we have just seen with telnet in your router. And this brings you to complete step 3.
If you do something wrong, the server will reply with an error message. When you receive the message END, you should always send BYE to let the server close the connection. Do not close the socket on your side without sending BYE first. The manager should die with no errors if you do things properly, when the last router sends BYE.

[Step 4 – Sending a message]

Now things are getting serious: it's time to send data messages. The TCP connection is not enough anymore. Each of your routers must implement a UDP server as well, listening on the port that is specified in the third parameter of the command line. Routers exchange messages using the UDP connections between each other. **It is strictly forbidden to exchange any message between nodes using other channels, files, or shared memory. Any attempt to do so will be considered cheating.**
A data message delivery is initiated by the manager. Our manager knows the UDP port each router is listening to, from the handshake. When a new message is generated, it is built and sent to the UDP port of the node marked as source in the XML file. The UDP message has a simple structure: the **first byte** is set to 1. The next **two bytes** represent the address of the destination. The rest of the message is the payload as specified in the <data> element. For example, the message described in this event:

```
<src>2</src>
<dst>3</dst>
<data>asdf</data>
```

is built as follows:



The first byte is always 1. As we will see later, your router instances must exchange control messages to collect information about the network topology beyond the one-hop neighbors. Since the only communication channel allowed is the UDP socket, nodes must have a way to determine whether a packet is a control packet, or if instead is a data message which must be forwarded. You can use the first byte to differentiate between messages the manager sends, for which the byte is set to 1, and the control messages that you will have to exchange among your routers. Using different values for the first byte you can define 254 different packet types, each with a different structure. This should be more than enough for any implementation design you choose. In fact, a practical implementation rarely uses more than 10 different packet types. The second and third bytes are the destination address for the packet, the remaining bytes in the packet are the payload. A packet size can be a lot larger than the 7 bytes shown in this example, but it will not be larger than what can be sent using a single UDP packet.

After sending the message to the source node, the manager patiently waits for your protocol to perform the routing decisions and deliver the message. The receiver (node 3 in this example) must inform the manager that the message was successfully received. This is simply done by sending the following line to the manager, using the TCP socket:

```
r: RECEIVED asdf
```

Where of course `asdf` must be substituted with the actual payload, which is different in each message.

In this first step routing choices are not necessary, since we are using src-dst pairs that are directly linked. The goal of this step is simply to verify that you are correctly keeping the neighborhood table as described during the handshake, and that a router can connect a deliver a message to one of its neighbors. For this step there is no need of control messages, although you are allowed to define and use some if you need to. The only thing your router should do is looking in the neighborhood list to find the hostname and the UDP port the destination node is listening on, then prepare a data message with the payload and send it. The destination should then inform the manager that the message was successfully received.

We can test this too before implementing it, once again using telnet and another really useful application: netcat. First of all run again the server, this time using the `Topology_and_message.xml` file. Then start the telnet clients, but do not initiate the handshake. The topology file that we are using sends two messages, both from node 1. Before we can start the simulation, we need to start another terminal and run the command:

```
nc -l –u udpport
```

This command starts a UDP server on the specified port, and will print the packets received. You will find `nc` very useful anytime you want to debug your networking code. For example, you can use it to start a server and verify what your program sends on the TCP session, if you have problems during the handshake.
Use the UDP port that you chose for node 1 during the handshake, since this is the only node that will receive UDP messages from the server in this simple experiment.
Now that your setup is complete, run the handshake on all nodes, and as soon as the last is done, you will see a message on your `nc` server. The first three bytes, a 1, a 0 and a 3 in this case, are not shown because they are not printable ASCII codes, but the payload, for which we intentionally used printable characters, should be there:

```
asdf
```

At this point, what your code is supposed to do is deliver the message to its intended destination. In this initial experiment using telnet we will pretend that this happened, and we will send the acknowledgment to the manager, from the node that is supposed to receive the message. You can see in the XML file that the first destination is node 3. Go then to the telnet terminal that emulates node 3, and send this line to the manager:

```
r: RECEIVED asdf
```

The manager checks that the acknowledgment comes from the correct node, and that the payload matches with the one that was sent. If these two conditions are true, the manager replies with the following message:

```
m: OK
```

It then proceeds with the new message delivery, in this case again the source node is node 1, this time the destination is node 2. You should see a new message on your netcat window, once again the node address is not printed, but the payload is:

```
fghj
```

And the manager then waits for the acknowledgment from node 2:

```
r: RECEIVED fghj
```

If the acknowledgment comes from a node that is not supposed to be the receiver, or if the message was already acknowledged the manager replies with:

```
m: ERR RECEIVED
```

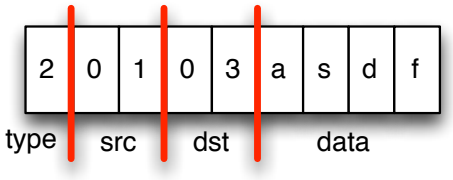
You will be penalized if any of these cases occurs.

If you did everything properly, you have reached the end of the simulation, the server sends you the `END` message, as before, and you can close the connection with `BYE`, as described before. It is time to implement this procedure in your router. You will create a UDP server in each router, which should listen on the port specified from the command line, and be able to reproduce the behavior described in this section. For now, all the destinations are one-hop, so all you have to do is follow these steps:

- when you receive the message on the UDP socket, first of all check the first byte to see if it is a 1 (this is a data message);
- now read the next two bytes of the message and find out which node is the intended receiver;
- now lookup on your neighborhood table. This is a one-hop neighbor, so the router MUST have received its address and port number during the handshake, and must have saved them in a apposite data structure;
- verify if the link is valid (i.e., the cost is not negative);
- now that you found the hostname and port on which the destination is listening, prepare a UDP datagram, put the message in its payload, and send it to the destination;
- finally, the destination router must send the RECEIVED message to the manager using the TCP socket, and wait for the OK from the server.

You have completed the next step, good job! **If you did not receive any error message from the manager, and instead the manager sent you the `END` message, you are have successfully earned full credit for this step.** We will not test with the XML file that we give you in the example; we will use a significantly larger network topology and number of messages. However, they will all be messages deliverable using one-hop routes, and we will not yet test links with negative values. You are encouraged to try many different topologies once you verify that you code works with the simple one that we give you.

Hint: You might want to differentiate between packets that the manager sends, and packets that the routers forward. You can do so by defining your own packet types, let's say that you decide to use 2 as the first byte for forwarded messages. The rest of the message, of course, can have a different structure as well. For example, you can add the source of the message:



To complete this task, you do not necessarily have to change the data structure, but we do not want to limit your creativity, we would like to see original solutions, and if you need to define different packet structures, feel free to do so.

[Step 5 – Logging Debug Information]

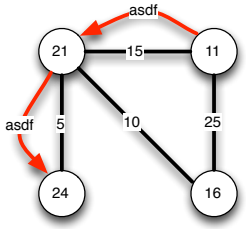
This step will be quick, but it is necessary to prepare the ground for your multi-hop routing implementation. All you have to do is implement a function in your router that after the handshake can be used, at any time, to send logging information to the manager. Some of these messages are used to grade your assignment by reconstructing your multi-hop routes and analyze their correctness. But you can also use the logging feature to debug your code. First of all, you need to signal the node that you are ready to start sending log messages:

```
r: LOG ON
m: LOG ON
```

This step must be completed only once on each client, right after the handshake, or in any case before sending the first message. When logging is on, the manager expects one log message from each of the nodes that forward a data packet to a neighbor. These log messages must be as follow:

```
r: LOG FWD dst data
```

where the `dst` field is the ID of the neighbor to which the data packet is sent to (not the final destination), and data is the content of the packet, in our example `asdf`. For example, this is the expected log messages to be sent in the hypothetical route:



Node 11 should send:
```
r: LOG FWD dst asdf
m: LOG OK
```

Node 21 should send:
```
r: LOG FWD 24 asdf
m: LOG OK
```

Node 24 should send:
```
r: RECEIVED asdf
m: OK
```

If node 24 sends the RECEIVED string before node 21 sends its LOG FWD message, there will be an error. To avoid this problem, node 21 should **first** send the LOG FWD message, and **then** send the message to the next node. Every message that starts with `LOG` is printed in a text file stored in the `log/` folder. One log file is created every time you start the manager. Each line of the log file starts with the ID of the node who sent the log message. You can use this logfile to debug your code, we will use the LOG FWD lines to reconstruct the path of your messages and grade your MP. But for now, all your router should do is delivery messages to its direct neighbors, using one-hop links. In the `Topology_and_messages.xml` two messages are sent, from 1 to 3, and from 1 to 2. Thus, this is the sequence of messages that the manager should receive (`r1` are messages sent by router 1, `r2` by router 2 and `r3` by router 3, `m:` are messages sent to either router from the manager):

```
r1: LOG FWD 3 asdf
m : LOG OK
r3: RECEIVED asdf
m : OK
r1: LOG FWD 2 fghi
m : LOG OK
r2: RECEIVED fghi
m : OK
m : END
r1: BYE
r2: BYE
```

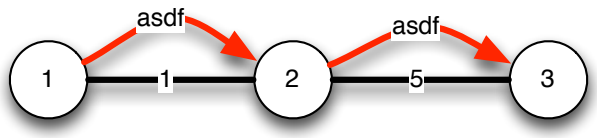
As soon as you implemented the log of forwarded messages you can consider this step completed. Additionally, you can send any other kind of log message to the manager. To grade step5 we will analyze the logs and verify ONLY that the correct number of FWD logs are received, and that the data matches with the packet payload that the manager sent. We do not evaluate the number or the format of any other log message that you send, this feature is provided only to support you in debugging your code, and you can leave it on in your submission too. Combining well designed XML test files and well planned logging should guide you through the next, and most crucial, steps.

[Step 6 – Multi-hop Routing]

We said a number of times that in this MP you will implement a routing algorithm, but so far all we have done was setting the stage. All your code can do at this point is keep track of the topology and send messages from a node to one of its neighbors. This is good, but not impressive. **You have worked in pairs so far, now it is time to work on the individual part of your assignment. Before continuing, if you haven't done it already, please read all the rest of this document, to find the necessary instructions on how to work on your individual part.**

This is the crucial point of your MP, here you are implementing the protocol that will enable multi-hop communication. We have seen a number of routing protocols in class. You can pick any of those, or design your own solution if you want to be creative. You will have to design your control messages to share topology information with the other nodes in the network, and you might probably want to change the format of the data messages as well, to keep track of a message route or append control information to data messages to save overhead. This is entirely up to you, and as long as the destination sends the RECEIVED message to the server, the routing algorithm is considered functional. **We remind you that no communication channel other than the UDP sockets is allowed. Any attempt of using files, pipes, shared memory or any other kind of IPC will be considered cheating.**

First, we will start with a basic case. We provide you with a very simple topology file, `Topology_and_messages_mh.xml`, which defines three nodes with two links, and one message sent over a two-hop connection as described in the picture:



The path is unique, so there is not much space for being creative. However, in order for the message to go from 1 to 3, the source node (1) must know that 3 is reachable through 2. This information is not provided during the handshake, since 3 is not a direct neighbor of 1. This means that your routing algorithm must implement some mechanism to discover the route between nodes that are not neighbors. This is the goal of a routing protocol. You can implement a procedure that periodically updates and distributes information about the topology, or rely on a proactive search for a path after receiving the data message, or a hybrid of the two techniques, or any other policy you can think of. **The only aspects of your implementation that we will evaluate are the following:**

- Messages eventually reach their destination.

- The message is not forwarded using links whose cost is negative (this means that the link is temporarily down).
- The FWD log messages, and the RECEIVED message allow us to reconstruct the path that your message took (i.e., no FWD message is missing, and they come in order). **This is very important, if your code works properly but at least one node does not use the LOG ON feature for the whole time, our autograder will consider your code non working.**

A message **cannot** be replicated in multiple copies; at any specific moment only one node should have it in its queue. When LOG ON is used, the manager would issue an error is the message is replicated.

If your routing algorithm causes the message to be forwarded in a loop once, you will not loose points for this step, as long as the loop is eventually detected and interrupted and the message is successfully delivered. An **undetected** loop (i.e., a loop that happens more than twice in a row) will instead be considered a failure. Loops are the nightmare of all routing protocols designers. But this mostly happens because of fluctuating links (link whose cost changes too fast.). In this case, with a static topology, your life should be a lot easier. But be careful and do a lot of test, as we said, because this is the core of your MP.

Hints:

- Clearly the topology file that we provide is too simple, and it only serves as starting point. You will have to implement something a lot more complex to test your algorithm, and we let you decide how to design your test cases. We will discuss later how to run multiple instances of your router without the need of opening one console per instance (a decent size topology will have 20 or more nodes).
- If you have not done it yet, you probably want to define a new packet type for control messages, which routers exchange among each others to perform the routing auxiliary functions that your protocol needs, however you can also consider combining the payload of a data message with control information in a single packet, and opportunistically distribute the control information without incurring the overhead of a new packet. In both cases, as long as the first byte of the packet differs from 1, you have absolute freedom, don't be afraid to be creative.
- Design your protocol carefully and think of all the possible scenarios before implementing it. Implementation can be tricky, and you want to have a clear idea of what your code should do before implementing it. Office hours are not just to help you fix segmentation faults or memory leaks. Come and talk to us about your design idea if you need to.
- In this step, we will test only a connected topology (i.e., there is at least one path between any pair of nodes). This means that every message should reach its destination. However, depending on how you design your protocol, some nodes might have a stale view of the topology and thus might not be able to find a proper route. This behavior is wrong and would cost you points, so you should fix it. But if you decide to give up on routing a message, and want to drop it and get the next one from the manager, you can do it. Let's assume that the payload for the current message is `asdf`, the node that last received the data message, must send the following command to the manager using the TCP socket:

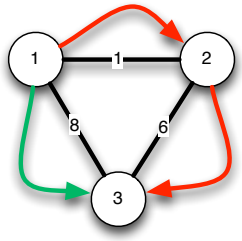
  ```
  r: DROP asdf
  ```

  The server replies with `OK` and proceeds with the next message. A dropped message is not as bad as a message that reaches one node using an invalid path, but you will not get full score. Essentially, we compute the fraction of sent messages that are dropped, and reduce your score for step 6 accordingly. If you drop 50% of messages, your score will be halved. If your messages instead goes from one node to another when there is no direct link among them, your score will be 0.

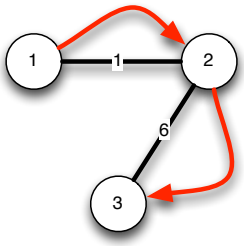
[Step 7 – Updating the Network Topology]

We are at a good point: your routers can successfully send messages over a multi-hop link, and you tested a good number of cases, from simple to more complex (although you are at this point working individually, you can share test cases and ideas with your partner). What we will do in this step is making sure that your code can react to changes in link topology during the simulation, using the topology file `Topology_and_messages_dynamic.xml`. Basically, as we have seen, each link has a cost. A negative cost, we know, means that the link is down, and no message should travel through that link. Step 7, as the previous one, will only evaluate the successful delivery of a message, the absence of undetected loops, the correct use of FWD and

RECEIVED messages to the manager, and the absence of transmissions over links that have a negative cost. Still at this stage, we will not grade the cost of your route compared to the optimal cost. The provided XML file starts with the triangular topology and a message being sent between node 1 and 3, for which two paths are acceptable:



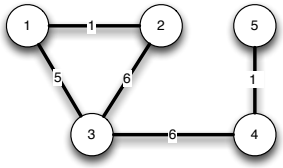
Your code can use the single 1→ 3 link, more expensive, or the cheaper two-hop link 1→2→3. This depends on the algorithm you designed, and both choices are fine. After the first message, a link cost is changed to negative, and the message is sent again. This time only one path can be used:



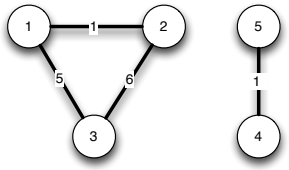
Hint: this is an exercise, and you have probably realized that we are giving you a simplified version of reality: events are generated in a sequence, no link cost changes while a message is in transit. This unfortunately is not true in real life, which makes a network programmer's life a living hell sometimes. But do not worry about this for the MP, we will not change link costs while the packets are in transit, although depending on how you implement your update procedures, some node far from the changed link might not know yet of the new link cost when a message is sent. We will tolerate a certain time between sending a link cost change message, and receiving the acknowledge from your routers, so if you want you can send the COST OK message only when you are sure that the network has been updated. However, if it takes more than 5 second to your code to return a COST OK message, our autograder will kill your program and consider the execution failed.

[Step 8 – Partitioned network]

It's time to test the last feature of a good routing algorithm: determining when a node is unreachable. A network partition happens when the nodes are divided in two subsets which are not connected. One example of a partition network with 5 nodes and four messages is provided in the file `Topology_partitioned.xml`. Initially the topology is this:



and three messages are sent, from 1 to 3, from 4 to 5, and from 5 to 3. Each of these messages should be delivered successfully as we have seen so far. After the three messages are delivered, an update in the cost of the link between 3 and 4, brings the link down, making the network partitioned:



One last message is sent again from 5 to 3. This time a path between the two nodes does not exist, and the manager should receive a message informing it that the message could not be delivered and has been dropped:

```
r: DROP partmsg
m: OK
```

Based on the routing algorithm that you decide to implement the message could be dropped immediately at 5, or could be forwarded to 4, which drops it after verifying that the link with 3 does not exist. Both solutions are correct, and on a general base, any node that has a multi-hop path to the source can be the one that drops the packet. The only two cases in which step 8 fails are when:

- the message is dropped by a node that should not have received the message in the first place (i.e., the message traversed an non-existing link before being dropped, or a FWD message was not sent to the manager);
- the manager receives a RECEIVED, DROPPED or FWD for a message after receiving a DROP (i.e., the message was not really dropped).

[Step 9 – Testing your router]

Congratulations. You have reached the end of this MP. Of course, all the test cases that we used so far are very simple and we used them just to help you understanding each step. It's now time to design your own XML files, with a combination of link costs changes, long multi-hop paths, and network partitioning and rejoins. For a good testing, you will need to define an XML file with at least 20 nodes, but you can go much higher than that, and you need a complex topology too. Of course, since each router simulates one node, you will need to run a number of instances equal to the number of nodes that you have in the topology file. Having 20 or more terminal windows open at the same time is inconvenient, you might find more useful to follow these instructions. First, select a random set of UDP ports for your routers to listen on, and create a file `ports.txt` listing each of the ports you intend to use exactly once. You will need a number of ports at least equal to the number of routers you want to start, but you can list a larger number, too. Then simply run the following command:

```
for i in `head -n 20 ports` ; do ./router1 localhost <port> $i &  done
```

where 20 is the number of routers that you need for a specific experiment (remember that it must be equal to the number of nodes in the topology), and of course router1 (or router2) is your executable. This command starts 20 instances of the router and executes them at the same time in one console. This of course means that their output would be mixed together, and this is why we suggest that unless strictly necessary, after the first development phase you avoid `printf`, and do all your debug by sending LOG messages to the manager. Now remember, your code will be tested against a number of different cases, please plan the implementation phase early, so that you are left with enough time to study complicate scenarios and solve the issues that will arise. Differently from the previous MP, here most likely the problems you will have to solve will be with the logic of your algorithm, more than with its implementation (e.g. loop detection, stale link costs). Try to design carefully the protocol before diving yourself in the coding madness.

NOTE: When you are done with your tests, you can kill all the routers using the following command:

```
killall -9 router1
```

Hand In

This MP is part a team effort, part individual. This time, you have complete freedom on how to structure your code, you can first work together on the same file (router.c) to complete the common part, then copy the file and work each on one copy, using different filenames. Or you can keep the same router.c file for the common part, and then

create two separate .c files to implement your individual part, using a common interface, and link the main file to the correct implementation at compile time. We really do not have any preference, although if you are in doubt you can come and talk to us during office hours. All we need is that your Makefile defines two targets, so that calling **`make router1`** and **`make router2`** create two binaries named respectively **`router1`** and **`router2`** which will be graded independently, resulting in two grades, one per each user. If you are working in a solo group, you must only implement **`make router1`** (not **`make router`**).

To help us in grading your MP and understanding who is responsible for **`router1`** and **`router2`** we will call your code using the following commands:

```
./router1 -netid
./router2 -netid
```

And we expect your code to print EXACTELY the following string:

`netID`: *yournetid*

(there is a space after the colon).
The two binaries will be graded independently and will result in two score reports, one per user, although the first steps should report the same score for both implementation. You must provide a README file which contains the following sections:

- A list of references to publicly available code, if any, that you used to write your solution.
- A description of the general architecture of the common part of your software. Did you use a multi-thread architecture to separate the UDP socket and the TCP socket? How are you synchronizing your threads?
- A brief description of how you split the work for the common part, did you work together or did you split the tasks? How did you organize your code? Whose code is the one compiled by **`make router1`** or **`make router2`**?
- A description of the protocols that each student implemented. If you decided to implement a protocol that we discussed in class, describe how it is linked to your common framework, what is the software architecture that you chose, what problems you had. If you decided to design your own protocol, provide enough detail on the routing decisions, the way loops are avoided or detected, the way partitions are handled, the way links updates are distributed, and anything that is necessary to understand your protocol implementation.
- Finally, describe what topologies you designed to test your protocol, and what results, expected or unexpected, you obtained.

The autograder needs a few information to understand your submission and grade it correctly: you must also prepare a file named **`files.txt`** in which all the files that matter to your submission should be listed. This includes all the source files or header files that you created, and the Makefile. You can create as many additional files as you need, but do not use additional folders.
Basically, by copying all the files listed in files.txt in a separate folder and running **`make router1`** or **`make router2`**, the compiler should be able to produce the two binaries.
This file is essential to the autograder, so please do not forget it or change its syntax. Simply list the files, with no directory information, one per line. Do not include the manager file, nor any subdirectory. We will not accept the submission of any binary or shared library, nor .o files. Your Makefile as usual must include a "clean" target, so that after running the command

```
make clean
```

none of the files generated by **`make router1`** or **`make router2`**, nor any other temporary file that your code creates remains in the folder. This excludes the logs generated from the manager, which can remain in the folder.

To hand in your homework, first add any new file you created and you want to submit by using the command:

```
svn add <filename>
```

This does NOT submit your homework, you need to follow the next step to commit it.
Once you are ready to submit your homework, type the following while in the directory containing the assignment:

```
svn ci –m “”
```

You may commit your MP as many times as you'd like. It's a great way to 'save' the work you've done so far. We will use the last submission you made for the MP when we grade your MP. Once a file has been committed to subversion, it has been submitted. You can verify the files on subversion by viewing your subversion though a browser, as explained for the previous MPs.

A file cannot be graded if it has not been committed. Failure to commit a file on time will result in your MP being considered late. It is your responsibility to ensure all your work is committed by the due date.

Grade

Your score will be assigned separately by running simulations designed to test each step. You will get partial credit for completing each step, and there might be a finer level of grading if one of your steps passes simple test cases but fails in more complex ones. **Memory correctness is graded ONLY if your code executes correctly Step 6.**

The Grade breakdown is the following:

[Step 1 – The TCP connection] 5%

[Step 2 – The Handshake] 10%

[Step 3 – The Link Cost Events] 10%

[Step 4 – Sending a message] 10%

[Step 5 – Logging Debug Information] 5%

[Step 6 – Multi-hop Routing] 15%

[Step 7 – Updating the Network Topology] 15%

[Step 8 – Partitioned network] 10%

[Valgrind correctness] 5%

[Valgrind leaks] 5%

[README] 10%