CMSC 417 Computer Networks	**Spring 2009**
Programming Assignment 3 — Distance Vector Routing	
Assigned: Feb. 25	*Due: Mar 11, 23:59:59*

1 Introduction

This assignment will implement distance vector routing. We will implement a *virtual* network on top of UDP. In this virtual network, UNIX processes will be network nodes, and links will be created using UDP. Since nodes in our virtual network are just processes, multiple nodes may reside on the same (physical) host. This also means that there are going to have to be as many instances of the program running concurrently as there are virtual nodes.

2 The Network and the Scenario File

The format to define our network[1] is specified using a scenario file. An example scenario file that (initially) defines the network shown in Figure 1 is shown in Figure 2. Note that virtual nodes 0 and 1 both reside on physical node `snickers`.

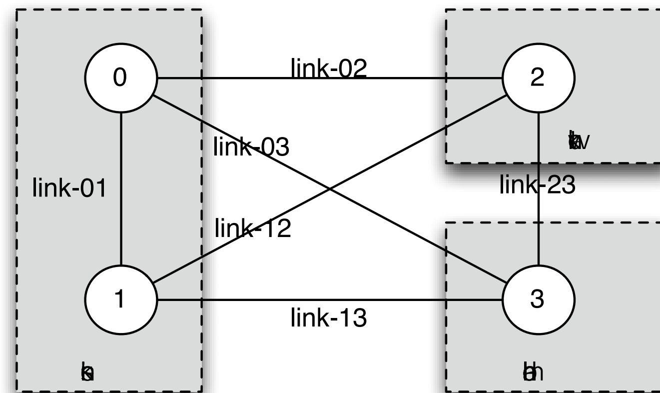

Figure 1: Initial virtual network configuration: Shaded rectangles correspond to physical nodes and the solid circles correspond to specific virtual nodes. All arcs represent virtual links.

The scenario file (Figure 2) maps to the network shown in Figure 1. The scenario file defines the set of nodes and a set of event sets. An event set consists of a set of events that affect the links in the network. There are three types of events: establishment of a link, tear-down of a link, and updating the cost of a link. All events in an event set are *executed sequentially without any delay.* How event sets are ordered is configurable — for this assignment, your task is to run the distance

Unless otherwise noted, we mean the virtual network when we say network.

vector algorithm for a fixed amount of time before executing events in the next event set. Thus, your code can be structured as follows:

```
boolean nextSET ← true ; nextSET is global
alarmHandler() {
      ; The alarmHandler is invoked every x seconds; x is a parameter
      nextSET ← true;
}
...
do
      if (nextSET is true)
            es ← ⟨ next event set ⟩
            ⟨ dispatch all events in es ⟩ ; This will cause changes to the links in the network
            nextSET ← false;
      ⟨ execute Distance Vector protocol ⟩
      ; You are required to use select() with a timeout in the DV protocol,
      ; rather than spinning inside the while-loop. (See Section 4.2)
while ⟨ there are more event sets to process ⟩
```

Obviously, you don't have to follow this blueprint, this is just one possible way.

2.1 Scenario file format

- The scenario file begins by listing all the virtual nodes in the network and may contain up to 256 virtual nodes. Virtual nodes are declared as follows:

  `node` ⟨ node-id ⟩ hostname

  The node id is an unsigned integer and corresponds to the virtual node identifier (must be unique) and the hostname is the host on which the process corresponding to this virtual node resides.

- After the virtual nodes are defined, the scenario file consists of a set of event sets. Event sets themselves consist of events and are delimited by "(" and ")". Thus, the rest of the scenario file looks like this:

  (⟨ set of events ⟩) . . . (⟨ set of events ⟩)

- There are three types of events in an event set. An event set may contain an arbitrary number of events of any given type in any given order. (Of course, the events must be consistent, i.e., an event cannot refer to a node or a link that does not exist.) Specifics of events are as follows:

  - The establish event establishes a new link in the network. The syntax is as follows:

    `establish` node ⟨ node-id ⟩ port ⟨ integer ⟩ node ⟨ node-id ⟩ port ⟨ integer ⟩ cost ⟨ integer ⟩ name ⟨ string ⟩

    This command will establish a link between the two nodes (and associated port numbers) whose node ids are specified. These nodes must already exist and the port numbers must not have been used before to define a link. The link has a cost given as an unsigned integer and a "name" specified as a string. All subsequent actions on this link will just use this string to identify the link. Hence, link names must be globally unique.

– The following event is used to tear down an existing link:

    `tear-down` ⟨ `string` ⟩

  Once again, the named link must already exist.

– Lastly, the cost of a link can be changed:

    `update` ⟨ string ⟩ `cost` ⟨ c ⟩

  The string should identify an existing link and the cost should be positive.

- A scenario file can also contain comments guarded by ";".

```
node 0 snickers ; Node numbers must be non-negative
node 1 snickers
node 2 twinkie
node 3 lemonhead
(
; Establish  a bunch of links --- this defines a ``virtual'' network
establish node 0 port 9001 node 1 port 9010 cost 10 name link-01
establish node 0 port 9002 node 2 port 9020 cost 11 name link-02
establish node 0 port 9003 node 3 port 9030 cost 12 name link-03
establish node 1 port 9012 node 2 port 9021 cost 13 name link-12
establish node 1 port 9013 node 3 port 9031 cost 14 name link-13
establish node 2 port 9023 node 3 port 9032 cost 15 name link-23
)
(
update link12 cost 999
update link13 cost 777
;
tear-down link-02
tear-down link-03
)
```

Figure 2: Sample Scenario File

2.2 Packets

Routes are disseminated using an advertisement packet with the following structure:

```
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    type       |   version     |          Num. Updates         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   dest_0      |           min_cost_0                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   dest_1      |           min_cost_1                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           .............                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   dest_n      |           min_cost_n                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Type – Set to 0x7 for this assignment.
- Version – Set to 0x1.
- Num. Updates – Number of distance vector pairs in this advertisement. This must be more than zero for all legal advertisements.
- Dest – Assume the advertisement is being sent from node a to node b and one of the routes being advertised is a route to node c. Then the Dest field corresponding to that particular route is c.
- Cost – Using the terminology from above, the cost field corresponds to the actual cost of the route to destination c as advertised by node a (to node b).

3 What We Give You

A substantial part of the project, including most of what is detailed below, will be given to you so you can concentrate on coding the distance vector part.

3.1 Parser

You will be given a `flex` and `bison`[2] parser which will parse the configuration file and automatically create a global virtual-node-to-hostname mapping and a two-dimensional local event structure that maps to the set of event sets. The interface is in the form of the `ruparse()` function. You must call the `parser_init` function before calling `ruparse` as shown below.

```
char *sc_file;
extern int ruparse();
int main (int argc, char *argv[])
{
    parse_arg(argc, argv);

    parser_init(sc_file); // sc_file contains the name of the scenario file
    ruparse();
    ........
  }
```

The `ruparse` function creates a two-dimensional event list. Each `column` in the 2-D event list corresponds to an event set in the scenario file. For example, in Figure 3, we see an event lists containing two event sets. In the first of these event sets, there are four `establish` events and two `tear-down` events. Since the events in a given event set can be considered to be occuring concurrently, this event set represents four links being established while two previously established links are being torn down. Note that once a process calls the parser using `ruparse`, it **never** has to bother with the scenario file again and it never has to call `ruparse` again, since all the information in the scenario file has been read into the event list. Note also that among the parameters to `parse_arg` are the command line arguments to your virtual node process. The parsing code uses this information to include in the event list only those events that are of interest to this particular virtual node. So, for example, if link-23 in Figure 1 is torn down, then this event will not show up in the event list for virtual nodes 0 or 1, since the link is not directly connected to either virtual node (of course, information about the link going down may eventually propogate to these virtual nodes via the routing protocol).

Each element of the event set is a `struct es` (struct event set). The definition of the event set is as follows:

---

[2] If you don't know anything about parsing, don't worry, you will not be required to do anything with `flex` or `bison` for this assignment.

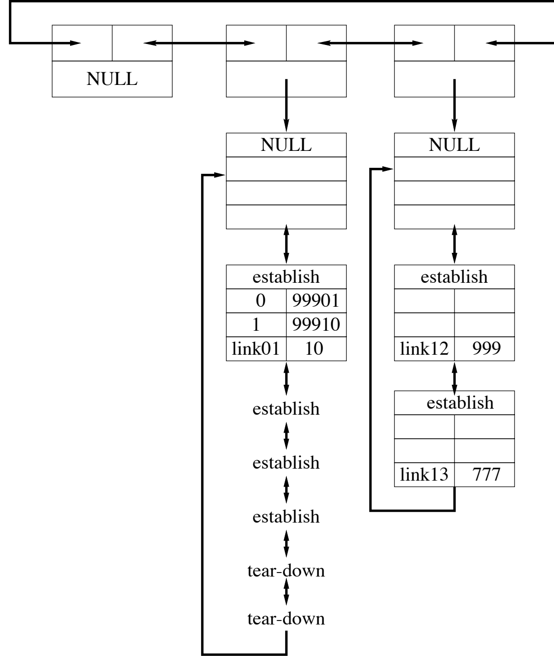

Figure 3: The two-dimensional event structure created by the parser. The top list is the global event list. Each element of the global event list is an event set. Each element of the event set is an event. All our queues start with a null element — thus, the global event list and each event set has a null element in the beginning.

```
struct es{
  struct es *next; // to create the 2-d list
  struct es *prev;

  e_type ev;       // ev is one of establish, tear_down or update
  int peer0, port0, peer1, port1;
  int cost;
  char *name;
};
```

Eventually, the virtual node will have to *dispatch* these events (i.e. it will have to send out routing advertisements that reflect the new network state). We discuss dispatching events in Section 3.4. The parser resides in `*ru*` files, and the event set is defined in the `es.[c|h]` files.

3.2 Nodes and Links

As the parser runs, it also creates a set of nodes to hostname mappings. This mapping can be accessed by using the `(char*) gethostbynode(int node)` function defined in `n2h*` files. The node id must be defined in order for `gethostbynode` to return anything meaningful.

During each dispatch of an event set (column), the local `link set` should be updated when necessary. Your routing algorithm then uses it to collect distance vector information, update its routing table. The local link set has the following structure:

```
struct link {
  struct link *next;  // next entry
```

```
    struct link *prev;  // prev entry
    node peer0, peer1;  // link peers
    int  port0, port1;
    int  sockfd0;       // if peer0 is itself, local port is bound
    int  sockfd1;       // if peer1 is itself, local port is bound
    cost c;             // cost
    char *name;         // name of the link
  };
```

The methods to access the link set are:

```
int create_ls();  // initialization

int add_link(node peer0, int port0, node peer1, int port1,
             cost c, char *name);

int del_link(char *n);
struct link *ud_link(char *n, int cost);  // update link
struct link *find_link(char *n);
void print_link(struct link* i);  // print info about a single link
void print_ls();  // prints entire link set
```

A virtual node process must call `create_ls` to initialize the link set at a virtual node. The `add_link`, `del_link`, `ud_link` functions mutate the link set. The `print_link` and `print_ls` print information about a given link or the entire set at the node.

Note well: When a link is added into the local link set, socket(s) corresponding to the link are **not** automatically allocated. You must write the code to associate the sockets yourself. Note that you can obtain a link structure using the `find_link` function, but you will have to know the name of the link.

Link sets are defined in `ls.*`.

3.3 Routing Table

We provide a set of routines to manipulate routing tables. The methods to maintain routing tables are:

```
int create_rt();
int add_rte(node n, cost c, node nh);
int update_rte(node n, cost c, node nh);
int del_rte(node n);
struct rte *find_rte(node n);
void print_rte(struct rte* i);
void print_rt();
```

The function `create_rt` must be called to create a routing table at a node. The functions `add_rte`, `update_rte`, and `del_rte` are used to add, update, and delete individual routing table entries. The logging functions `print_rte` and `print_rt` print individual table entries and the entire table, respectively. The function `find_rte` is used to find an entry for a specific destination.

3.4 Dispatching Events

After the event list is created, a virtual node has to *dispatch* functions for each event in the event sets. As we said before, all the functions in a single event set will be executed sequentially. (Note again that the event set at a virtual node will only contain events that pertain to this virtual node — events at remote nodes that are in the scenario file are not added to the event set at the local node). The event set code defines the `walk_el` function that traverses the event list and the `dispatch_event` function that modifies the link set as appropriate.

4 The Executable

4.1 Command Line

eccmdline Your executable should take in the following command-line options:

```
rt -n <node_id> [-f <scenario_file>] [-u update-time] [-t time-between-event-sets] [-v]
```

The `-n` option is mandatory and specifies the node id that this process corresponds to; the optional `-f` parameter specifies a scenario file (default `config`), and the optional `-t` parameter specifies how long to wait between executing event sets (default 30 seconds). The `-u` option specifies how long to wait (in seconds) between sending out distance vector updates; this should default to 3 seconds. While the process is waiting, it should be receiving other updates.

If the `-v` (verbose) option is not present, your code should print the routing table to `stdout` after each event set is executed. Further, your code should also print each event in the event set that it acts upon and each routing update that changes the routing table.

If the `-v` option is present, the routing tables should be dumped after each routing update (whether or not it changes the routing table entries), in addition to the printing of events and updates as described above.

4.2 Requirements

There are two specific requirements for this assignment:

- You are required to use the `select` system call to multiplex reading from multiple descriptors. Since your virtual process will have multiple links incident upon it, it can receive a message from any link. If the node just does a `read` or `recvfrom` from any link, the process will be blocked till something actually arrives on that link. The UNIX (system) call `select` allows you to wait on multiple descriptors, and you should use this facility to implement your virtual node.
- Use the following port range. Suppose your account id is `cs4170xx`, the ports you should use are `1xx00 - 1xx99`. Thus, if your login id is `cs417008`, you should use port range `10800 - 10899`. Your final program must work with *any* scenario file, but when *you* are testing your project you should only use the ports you have been allocated.

4.3 Hints

- Unlike the previous assignments, a lot of the implementation specifics are up to you. This means you should plan them out before you start coding.
- You will have to add or modify several functions in several files, although probably not the driver/`main()` function.
- A good place to start would be in the `walk_el()` function or the `bind_port()` function.
- There is a lot of extraneous output in the code that has been given to you. This is for your initial benefit, but should be stripped out later.
- Obviously, all processes must be using the same config file.
- You don't have to worry about node discovery, all nodes are initially in the routing table.
- Don't worry about the counting-up-to-infinity problem for now.
- You will probably want to run a script to start all your virtual nodes at about the same time. The following script starts 3 nodes (node ids = 0, 1, 2) on the same physical machine, with the output directed to the files out.{0, 1, 2}.

```
#!/usr/bin/perl
foreach $i (0..2) {
    if (!fork()) { #child
        `./rt -n $i -f config > out.$i`;
        exit;
    }
}
```

5 Bonus

The bonus involves implementing a simple traceroute mechanism, which is used to determine the path that a packet is routed by your virtual nodes. For example, we may trace the path between node 0 (source node) and node 5 (destination node), and the result may look like 0→3→2→8→5. You will have to build a client that initiates the traceroute operation, and modify your virtual node to support this operation.

5.1 Traceroute Packet Format

The traceroute mechanism uses the following packet format (in UDP):

```
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type       |   Version     | Dest. Node Id |  Num. Visited |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Client's IP address                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Client's Binding Port      |   Visited_0   |   Visited_1   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    ....................                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Type – Set to `0xc0` for traceroute packet. You can use this field to distinguish between a traceroute packet and a route advertisement.
- Version – Set to `0x1`.
- Dest. Node Id – The destination node id for this traceroute operation. When routing the packet, each node decides the next hop based on this field.
- Num. Visited – Number of visited nodes.
- Client's IP address – The IP address of the traceroute client (In numerical format).
- Client's Binding Port – The port number on which the traceroute client binds and expects to receive a responding traceroute packet. It is randomly chosen by the client among non-priviledged ports. (Choose within the port range that is allocated to you. See Section 4.2.)
- Visited – Each byte stores the node id that the traceroute packet has visited. The node ids should be stored in order, where the node visited earlier should be stored before the one visited later.

5.2 Executables

You will have to provide a client called `tracert` that can run separately from your virtual nodes. The `tracert` accepts the following command-line arguments:

```
tracert -h <node_hostname> -p <node_traceroute_port> -d <destination_node_id>
```

The `-h` option specifies the physical hostname of the traceroute source node. The `-p` option specifies the port that the source node is expecting traceroute packets. This value should correspond to the `-p` option of the virtual node (see below). The `-d` option specifies the destination node id (we don't have to know the physical address and port of the destination node). Running `tracert` will output the path (in terms of node ids) from the source node to the destination node.

You will also have to add an option to the virtual node `rt`:

```
rt {options_listed_in_Sec_4.2} -p <traceroute_port>
```

The `-p` option specifies the port number that the virtual node will be receiving traceroute packets from the `tracert` clients.

5.3 Procedure

The traceroute operation acts as follows:

- The client sends an traceroute packet to the source node, with appropriate Dest. Node Id and the Num. Visited set to 0.
- On receiving a traceroute packet, each node takes the following actions:
  - Appends its node id to the Visited field of the packet.
  - If it is the destination node, sends the packet back to the client immediately. The hostname and port of the client can be found in the packet.
  - If not, looks up its routing table and relays the packet to the next hop.

5.4 An Example

Figure 4 shows an example of traceroute operation. First, the `tracert` client sends an empty traceroute packet to the `traceroute_port` of Node 1, specifying the destination as Node 3. Upon receiving this packet, Node 1 appends its id to the packet, and looks up what's the next hop to reach Node 3. It then relays the packet to Node 2. Similarly, Node 2 appends its id to the packet and relays it to Node 3. Since Node 3 is the traceroute destination, it sends the packet directly back to the client.

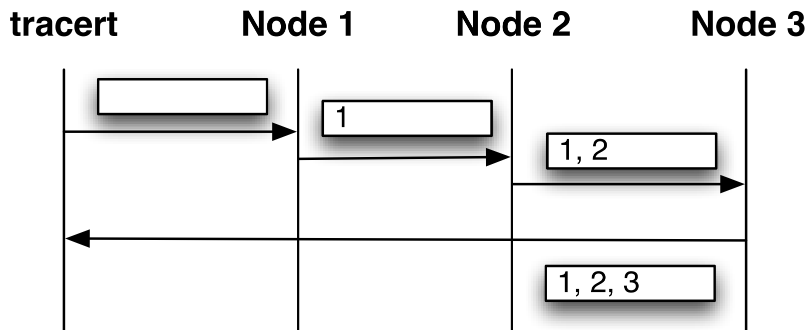

Figure 4: An example traceroute operation

5.5 Notes

- You don't have to implement any retransmission mechanism in the client.
- If there are more than 255 hops involved in routing a traceroute packet (e.g., in the case where there are routing loops), you may consider dropping that packet since 255 is the maximum number that can be represented by Num. Visited field.
- Make sure your virtual nodes and the DV algorithm are functioning properly before you work on the bonus part. If we can't verify your traceroute results due to the failure of your virtual nodes, you will receive little or no partial credit for the bonus part.

6 Submission

- Please submit your code as described in the forum.
- What to turn in: All of your source code along with a Makefile. Your Makefile must produce executable named `rt` after a single `make` command. If you want to turn in the bonus part, your Makefile should also produce a `tracert` executable.