CMSC 417 Computer Networks		**Fall 2022**
Programming Assignment 2		
Assigned: September 26	*Due: October 10, 11:59:59 PM.*	*Weight: 2.0x*

0 Updates

Route Dissemination Packets section has been updated. See the diagram on Section 2.2 for details.

1 Introduction

In this assignment you will implement distance vector routing. You 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.

The format to define our network is specified using a scenario file. An example scenario file and associated network is shown in Figure 1. Since nodes in our virtual network are just Unix processes, multiple nodes may reside on the same (physical) host. This is shown in Figure 1 — virtual node 0 and 1 both reside on physical node `fireball`.

1.1 What is in a scenario file?

The scenario file defines the set of nodes and a set of event sets. An event set consists a set of events that affect the links in the network[1]. There are three types of events: establishment of a link, tear-down of a link and updating the of 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 vector algorithm for a fixed amount of time before executing events in the next event set. We provide pseudocode in the project files to give you an idea of how to structure your code. However, you are not required to follow them.

2 Implementation

2.1 Scenario Configuration File

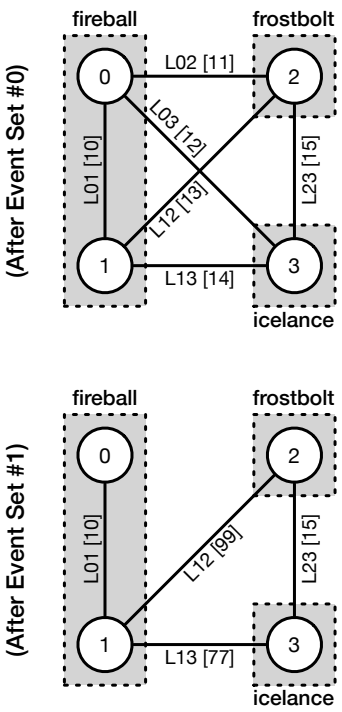
The format of the scenario file is as follows:

- 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.

[1] Unless otherwise noted, we mean the virtual network when we say network



(Scenario Configuration File)

```
; List of nodes
; Format: "node <id (>= 0)> <string>
node 0 fireball
node 1 fireball
node 2 frostbolt
node 3 icelance
; List of event sets. Each event set is enclosed in ( )
; Event Set #0
; Establish a bunch of links which defines a "virtual" network.
(
; Format: "establish node <id> port <int> node <id> port <int>
;          cost <int> name <string>"
establish node 0 port 01 node 1 port 10 cost 10 name L01
establish node 0 port 02 node 2 port 20 cost 11 name L02
establish node 0 port 03 node 3 port 30 cost 12 name L03
establish node 1 port 12 node 2 port 21 cost 13 name L12
establish node 1 port 13 node 3 port 31 cost 14 name L13
establish node 2 port 23 node 3 port 32 cost 15 name L23
)
; Event Set #1
; Updating link costs and permanently tearing down links
(
; Format: "update <string (link name)> cost <int>"
update L12 cost 99
update L13 cost 77
; Format: "tear-down <string (link name)>"
tear-down L02
tear-down L03
)
```

Figure 1: Sample scenario configuration file and the two relevant states of the network after the respective event set is executed. Shaded rectangles correspond to physical nodes and the solid circles correspond to specific virtual nodes. All lines represent virtual links between nodes, which are labeled by their link name and cost.

- 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: following following

`tear-down` ⟨ `string` ⟩

Once again, the named link must already exist.

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

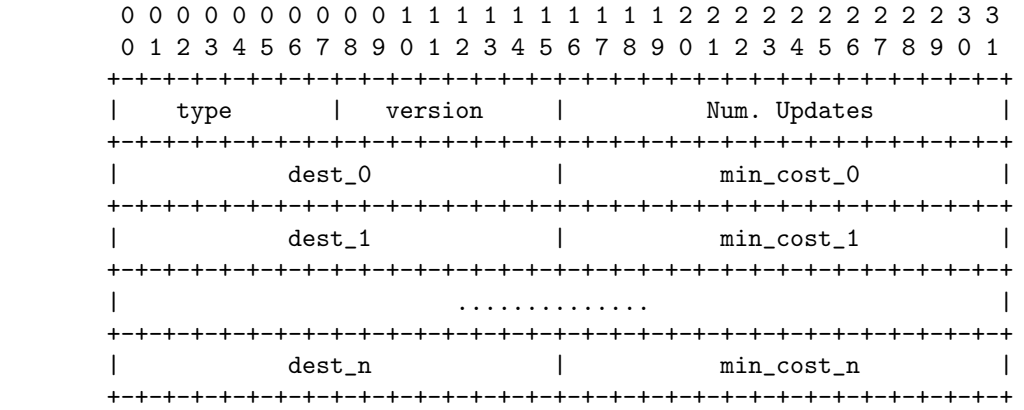
`update` ⟨ string ⟩ `cost` ⟨ integer ⟩

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

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

2.2 Route Dissemination Packets

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



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 from node a to node b and the `Dest` field is c. Node c is the final destination **to** which node a is advertising the min. cost to node b through node a.

Cost: Using the terminology from above, the cost field correspond to the actual cost of the route to destination c as advertised by node a (to node b).

2.3 Source Code

A substantial part of the source code will be given to you so you can concentrate on developing the distance vector part. This code can be found in your 'assignment2' repository, which can be found on gitlab:

git clone git@gitlab.cs.umd.edu:cmsc417-bobby-f22/students/⟨YOUR_DIRECTORY_ID⟩/a2.git

You are required to use either the `select` or `poll` system calls 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) calls `select` and `poll` allow you to wait on multiple descriptors, and you should use this facility to implement your virtual node.

Make sure you take enough time to familiarize yourself with existing code before starting! README.md in the repository root will contain additional information about the codebase itself.

2.3.1 Parser

You will be given a `flex` and `bison`[2] parser which will parse the configuration file and automatically create a global 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 a event set in the scenario file. Note that once you call the parser using `ruparse`, you **never** have to bother with the scenario file again and you never have to call `ruparse` again. All the information in the scenario file has been read into the event list.

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

```
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, you will have to *dispatch* these events. We discuss dispatching events in Section 2.3.4. The parser resides in `*ru*` files, and the event set is defined in the `es.[c|h]` files.

[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.

2.3.2 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.

In each dispatch of an event set (column), the `local link set` should be updated when necessary. Your routing algorithm then use 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 peer;  // the other node this one is connected to
  int  host_port, peer_port;
  int  sockfd;        // socket for the link. Bound to host_port
  cost c;             // cost
  char *name;         // name of the link
};
```

The methods to access the link set are:

```
int create_ls();  // initialization

int add_link(int host_port, node peer, int peer_port,
             cost c, char *name);

int del_link(char *n);

int ud_link(char *n, int cost);

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
```

You must call `create_ls` to initialize the link set at a 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.

Link sets are defined in `ls.*`.

2.3.3 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 .

2.3.4 Dispatching events

After the event list is created, you have 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 that the event set at a node will only contain events that pertain to this 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.

3 Command-line options and logging

You executable should take in the following three 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; 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) before sending out distance vector updates; this should default to 3 seconds.

Your code should print the following to `stdout`: 1) each event in the event set that it acts upon using `print_event`, 2) the routing table entry after it was changed in response to processing a routing update using `print_rte`, and 3) the full routing table after dispatching the entire event set using `print_rt`.

The `-v` option is optional for this project. However, it is recommended that you implement it for debugging purposes. If you choose to implement it, then the full routing table should be printed after processing every routing update (whether or not it changed any entries) using `print_rt`. You may have it print any additional information you find useful.

4 Additional Requirements

1. Your code must be submitted as a series of commits that are pushed to the origin/master branch of your Git repository. We consider your latest commit prior to the due date/time to represent your submission.
2. You must provide a Makefile that is included along with the code that you commit. We will run 'make' inside the 'assignment2' repository root, which must produce a 'rt' also located in the 'assignment2' repository root.
3. You must submit code that compiles in the provided VM or Docker, otherwise your assignment will not be graded.
4. Your code must be -Wall clean on gcc/g++ in the provided VM, otherwise your assignment will not be graded. (Note that bison may generate code that is not -Wall clean. This is fine.) Do not ask the TA for help on (or post to the forum) code that is not -Wall clean, unless getting rid of the warning is the actual problem.
5. You are not allowed to work in teams or to copy code from any source.