CMSC 417 Computer Networks Spring 2000

Project 4 — Distance Vector Routing

Assigned: February 29 *Due: March 9th, Midnight.*

1 Introduction

This assignment 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 that (initially) defines the network shown in Figure 1 is shown in Figure 2. 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 and Figure 2 — virtual nodes 0 and 1 both reside on physical node `poirot`.

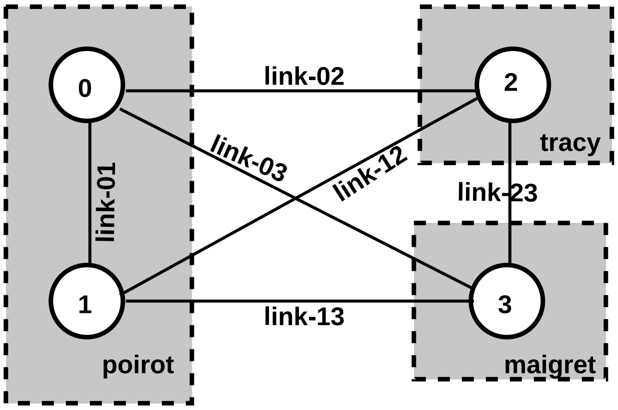

Figure 1: Initial virtual network configuration: Shaded rectangles correspond to physical nodes and the solid circles correspond to specific virtual nodes. All arcs represent virtual links.

The scenario file (Figure 2) maps to the network shown in Figure 1. It fills in the details needed to create each link. The details of the scenario file are postponed till the next section.

```
; Comments are guarded by ``;''
;
node 0 poirot ; Node numbers must be non-negative
node 1 poirot
node 2 tracy
node 3 maigret
;
;
; Event sets are enclosed in () parens
(
; establish link command format:
; establish node <id> port <int> node <id> port <int> cost <int> name <string>
; Establish  a bunch of links --- this defines a ``virtual'' network
establish node 0 port 01 node 1 port 10 cost 10 name link-01
establish node 0 port 02 node 2 port 20 cost 11 name link-02
establish node 0 port 03 node 3 port 30 cost 12 name link-03
establish node 1 port 12 node 2 port 21 cost 13 name link-12
establish node 1 port 13 node 3 port 31 cost 14 name link-13
establish node 2 port 23 node 3 port 32 cost 15 name link-23
)
;
(
; update link command format:
; update <link-name-string> cost <int>
update link12 cost 999
update link13 cost 777
;
; links can also be permanently torn down
; tear-down link command format:
; tear-down <link name string>
tear-down link-02
tear-down link-03
)
```

Figure 2: Sample Scenario File

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 of 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 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. Thus, your code can be structured as follows:

; `alarmHandler` *is called every timeout number of seconds*

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

```
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 ⟩
while ⟨ there are more event sets to process ⟩
```

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

2 Implementation

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.

- 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 ";".

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

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).

2.1 What we give you...

A substantial part of the project will be given to you so you can concentrate on coding the distance vector part. 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 individual port numbers assigned previously. Given that your class account login id is `[lg|sb]417xy`, the ports you can use are `a0xyn` where `n` ranges between 0 and 9 (inclusive). The value of `a` is 1 if you are in Section 0101 and 2 if are in Section 0201. Thus, if your login id is `lg41742`, you should can use port range 1042[0-9]; if your login id is `sb41742`, you will use port range 2042[0-9]. Your final project must work with *any* scenario file, but when *you* are testing your project you should only use the ports you have been allocated.

2.1.1 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();
    .......
```

[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.

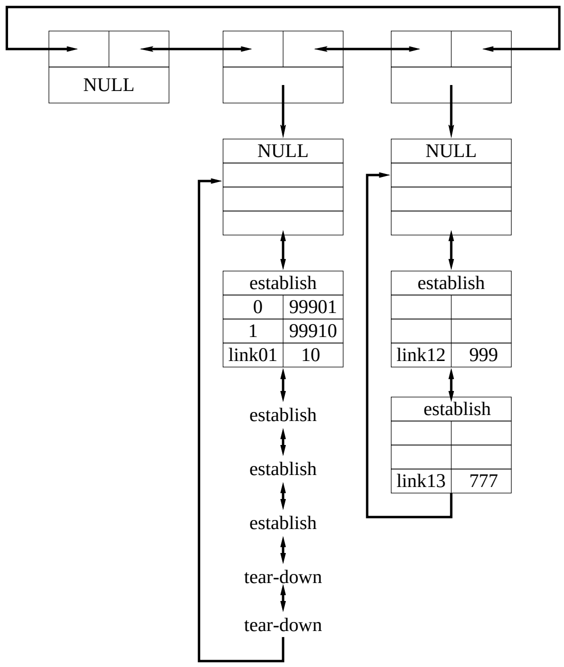

Figure 3: The two-dimensional event structure created by the parser.

```
  }
```

The `ruparse` function creates a two-dimensional event structure. Each `column` in the 2-D event structure corresponds to an event set in the scenario file. For example, in the sample scenario file (in Figure 2), there are two event sets and the event structure will look like Figure 3. Note that, the event structure in Figure 3 does *not* correspond to the two event sets in Figure 2. Also 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 structure
  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 2.1.4. The parser resides in `*ru*` files, and the event set is defined in the `es.[c|h]` files.

2.1.2 Nodes and Links

As the parser runs, it also creates a set of node-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, and 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 functions 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);

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 add a link, delete a link, and update the link cost, respectively (i.e., modify the link set). The `print_link` and `print_ls` print information about a given link or the entire set at the node.

Note that: When a link is added into the local link set, socket(s) corresponding to that 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.*`.

2.1.3 Routing Table

We provide a set of routines to manipulate routing tables. The functions 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 functioncreate_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 (*you should never have to use this function*).

2.1.4 Dispatching events

After the event structure is created, a 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 that 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 structure and the `dispatch_event` function that modifies the link set as appropriate.

3 Command-line options and logging

Your 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.

If the `-v` (verbose) option is not present, your code should print the routing table to `stdout` after each event set is executed. Furthermore, your code should also log 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 and every routing update (whether or not it changes the routing table entries).

In either case (i.e., with or without the `-v` option), all printing should be done using the print functions described above.