Computer Network Assignment 3 Write-up

Lucas Brasilino
lbrasili@indiana.edu

Sirshak Das[*]
sirdas@indiana.edu

1 Introduction

The third programming assignment from P358 Computer Networks class is to implement a Distance Vector routing. Each node, which are routers, has to send their cost for reaching all nodes within the network to its immediate neighbors.

The resulting Routing Information Protocol (RIP) service from this assignment might support update messages. These messages might be sent in a periodic and asynchronous[1] ways. The protocol is well defined in Figure 1 and must use UDP. After receiving update messages, a node uses Bellman-Ford Algorithm to update its routing table. Finally, if a node updates any costs in its table, it sends an update message to its immediate neighbors.

```
                              32 bits
+---------------------------------------------------------------+
|                        IP address (4 bytes)                   |
+---------------------------------------------------------------+
|                           cost (4 bytes)                      |
+---------------------------------------------------------------+
|                        IP address (4 bytes)                   |
+---------------------------------------------------------------+
|                           cost (4 bytes)                      |
+---------------------------------------------------------------+
                                ...
```

Figure 1: Update message

2 Implementation

The developed program, called `rip`, was written in C using a private repository in IU's GitHub server. It is organized in 3 directories:

- `src`: stores the source code files (`*.c`)
- `include`: stores the header files (`*.h`)
- `test`: stores sets of configuration files (`*.conf`)

Some design decisions were made for `rip` development. First, it is a multithreaded application. Second, internally not only node's IP addresses are stored and manipulated, but also the node's hostname. This fact enables more flexibility on the configuration file and better routing table printouts. Finally, the functions in source code follows the convention `rip_<file>_<name>()`, are organized in such a way that functions starting with `rip_<file>_*()` are placed in `rip_<file>.c` and each filename embed its purpose, as described bellow:

- `rip_main.c`: It is the application entry point (function `main()`) and the main process. Also contains routines for parsing the command line arguments and configuration file, for the distance vector graph initialization and for running the mainloop.
- `rip_net.c`: Encompass all network API, including the **send_advertisement** component (`rip_net_send_advertisement()`).
- `rip_obj.c`: This file contains all memory allocation deallocation functions for creating objects, like node information, route table entries, and protocol message entries.

[*]Names placed in alphabetical order
[1]As soon as there is some routing table update

- `rip_routing.c`: This file contains all the routing APIs including filling up of graph structure, distance vector and routing table.
- `rip_up.c`: Contains the entry points for the other two threads: **rip_up** and **rip_up_ttl**.

3 Compilation

On top directory a `Makefile` is provided for a easy compilation. As expected, packages like gcc, make and glibc-dev must be installed in the Linux system.

```
$ make clean && make
```

At the end of compilation process, the `rip` binary will be available in this directory.

4 Running the code

In order to rule the application's behavior, a set of command line options were defined. We have used the GlibC's getopt API to accomplish this functionality. The options are:

- `-c` ⟨`config_file`⟩ : points to config file
- `-i` ⟨`interface`⟩ : inform by which interface the process will bind to. The IP of it is also used as a identifier within the routing table (array `routingtable[]`) (Default: eth0)
- `-u` ⟨`port`⟩ : inform which UDP port might be used for receiving update messages
- `-p` ⟨`period`⟩ : period, in seconds, for sending update messages. (Default: 30)
- `-t` ⟨`ttl`⟩ : the expiration TTL from routing table (Default 3). The final TTL in seconds is calculated by period x ttl, so the default TTL, in seconds, is (3 x 30 = 90)
- `-y` ⟨`infinity`⟩ : set the infinity value (Default: 16)
- `-s` : enables split horizon
- `-d` : enables debug, so the `stderr` is sent to console, otherwise it will be sent to file 'rip_⟨`hostname`⟩.log'

5 Configuration file format

The configuration file follows exactly the formatting proposed by the assignment. The first column is a node identifier, which can be an IP address or its valid FQDN. In this context, *valid FQDN* means that it can be resolved using DNS or `/etc/hosts`.

Figures 2 and 3 shows examples of configuration files.

```
192.168.1.2 yes
192.168.1.3 yes
192.168.1.4 no
```

Figure 2: IP-based configuration file

```
basalt.soic.indiana.edu yes
blesmol.soic.indiana.edu no
bobac.soic.indiana.edu yes
```

Figure 3: FQDN-based configuration file

6 Application Internals

This section documents all application internals aspects.

6.1 Principal data structures

The very basic data structure is the `node_info_t`. It holds the FQDN name of the node and its IP address in a `struct sockaddr_in` structure.

```c
/**
 * Information about nodes (to be used in graph/table)
 *
 */
struct _node_info
{
    char *name;          /**< Name of the node */
    struct sockaddr_in *inet;   /**< Inet related information */
};

typedef struct _node_info *node_info_t;
```

The `route_entry_t` data structure uses the `node_info_t` information to maintain a routing table entry. The route table itself is the `routingtable[]` array. The global variable `rip_routing_table_entry_number` register the number of entries in the routing table.

```c
/**
 * Route entry for routing table
 *
 */
struct _route_entry
{
    node_info_t destination;    /**< Destination */
    node_info_t nexthop;    /**< Next hop */
    cost_t cost;        /**< Cost */
    unsigned short int ttl; /**< TTL */
};

typedef struct _route_entry *route_entry_t;

unsigned int rip_routing_table_entry_number;
route_entry_t routingtable[MAXROUTE];
```

As will be described later, the `rip` application uses a data structure called `adtable` to pass the advertised table received by the main thread (from a node) to the **`rip_up`** thread in order to do routing calculations. So, this data structure is type `advert_entry_t`:

```c
/**
 * Advertisement received from neighbor. This struct is
 * populated from message. advert_entry_t is not a pointer
 */
struct _advert_entry
{
    node_info_t neighbor;       /**< Advertisement sender */
    route_entry_t neightable[MAXROUTE]; /**< Advertised table */
    bool_t ready;                       /**< Advertised message ready for thread*/
    bool_t is_empty;                    /*is advent_entry buff empty*/
};

typedef struct _advert_entry advert_entry_t;

advert_entry_t adtable;
```

The UDP message itself must be in the form as depicted in Figure 1. To accomplish this the type `message_entry_t` has two fiels: dest_addr and cost, both 4 bytes legth.

```c
/**
 * Each entry of a message.
 * It is not a pointer because its desirable to be in-memory aligned.
 */
struct _message_entry
{
    struct in_addr dest_addr;
    cost_t cost;
};

typedef struct _message_entry message_entry_t;
```

The graph is implemented by `r_graph`, which is an adjacency table: 2 dimensional array, where each element has `cost` and `ttl` information.

```
typedef struct _route_graph_entry
{
    cost_t             cost;
    unsigned short int ttl;
} route_graph_entry_t;

typedef route_graph_entry_t*  route_graph_row_t;
typedef route_graph_entry_t** route_graph_t;
route_graph_t r_graph;
```

Finally, the distance vector is maintainend in `dist_hop_vect`, which is typed as `route_dist_hop_vect_t`, as an array with `hop_index` and `cost`. `hop_index` is the index of the nexthop node present the same array. This index is used to fill in the next_hop node details in the routing table. This is an index used also in the rows and columns in the graph.

```
typedef struct _route_dist_hop
{
    cost_t       cost;
    unsigned int hop_index;
} route_dist_hop_t;
typedef route_dist_hop_t* route_dist_hop_vect_t;

route_dist_hop_vect_t dist_hop_vect;
```

6.2 Main process and threads

The `rip` application runs a main process and spawns 2 threads: **rip_up** and **rip_up_ttl**.

The main process It parses the command line options, parses the config file, builds the initial routing table, starts to listen to UDP messages and spawns the **rip_up** and **rip_up_ttl** threads. After that, it enters in a infinite loop (`rip_main_loop()`) listen to other's node advertisements (`rip_net_recv_advertisement()`) and passes the received information to **rip_up** (`rip_obj_push_recv_advertisement()`) using the `adtable` data structure, using `lock` mutex for synchonization.

rip_up This thread retrieves the message received by the main thread and fill the advertised route into the graph structure. Then runs Bellman-Ford (`rip_routing_bellman_ford()`) on the graph which relaxes all the edges and computes the distance vector hold on `dist_hop_vect` data structure. This is a scaled down version of the `routingtable[]`, i.e, it only contains the nexthop as hop_index and the cost. Ttl is determined from the graph or from the message received. Once Bellman-Ford computes the distance vector, this thread compares the distance vector to the routing table. If the there is a change and sends a triggered update immediately. If not, it checks for the last time when last update was send and, if a periodic update is required, it sends accordingly. This thread also calculates the Convergence Time, by calculating the time elapsed from `start_time` to the time when no updates force a change in `routingtable[]`. When it occurs, the thread updates the start_time to the new time. This thread uses 2 mutexes: `lock` and `graph_lock`. The former for accessing `adtable` and the latter for accessing `r_graph`. It shares the first mutex with main process and the second one with **rip_up_ttl** thread.

rip_up_ttl : This thread sleeps for period time (defined by the `-p` option), wakes up, executes a procedure, and then goes back to sleep again. The procedure is to decrease the ttl value in each graph node and if ttl value reaches 0 it sets the COST to `COST_INFINITY` (16). It uses the mutex `graph_lock` to synchronize access to shared data structure `r_graph`.

6.3 Global, shared, conditional variables and mutexes

In the application implementation, all global variables are also shared at least between the main process and the thread **rip_up**. They are:

- `routingtable`: the routing table array;
- `rip_routing_table_entry_number`: the quantity of entries in routing table;
- `adtable`: the advertised table received from a neighbor;
- `r_graph`: the graph, i.e., the cost and ttl to a given node. Each index is associated with the index in the routing table;

The conditional variables are members of `adtable` data structure. They are:

- `adtable.ready`: Is used by the main process to signals **rip_up** thread that it must execute;

- `adtable.is_empty`: Tells **rip_up** thread if `adtable` is empty. If it is empty, **rip_up** checks if sending update is required (`rip_util_is_update_required()`). If the result is true, `rip_net_send_advertisement()` is called. If `adtable` is not empty, i.e., the node has received an advertisement message, it calculates and updates the distance vector and the graph.

6.4 Mutexes

Two mutexes are used in the application:

```
/* Mutex */
pthread_mutex_t lock;
pthread_mutex_t graph_lock;
```

`lock` mutex controls the access to `adtable` between the main process and **rip_up** thread. `graph_lock` mutex controls the access to `r_graph` between **rip_up** and **rip_up_ttl** threads.

7 Test cases

For application evaluation, some test cases were designed and performed.

7.1 Test Case 1 - Acyclic graph with single failure

In this test case, three nodes were executed in a linear topology as depicted in Figure 4. Table 1 shows the convergence time without using split horizon, while Table 2 shows the results with split horizon. In both tables the results are grouped by parenthesis, but each value correspond to a node. Finally, all times are in seconds, including Infinity and Period.

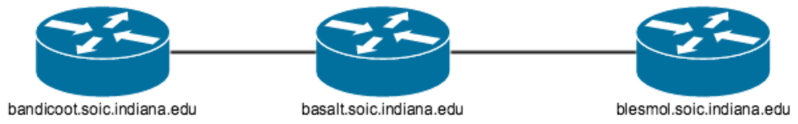

Figure 4: Test Case 1 three nodes linear topology

Table 1: Convergence time without split horizon

Infinity	Period	Initialization (bandicoot, basalt, blesmol)	One down (bandicoot, basalt)
16	10	(20, 9, 17)	(40, 60)
16	30	(61, 29, 54)	(90, 150)
50	10	(20, 9, 18)	(30, 50)
50	30	(60, 29, 57)	(90, 150)

Table 2: Convergence time with split horizon

Infinity	Period	Initialization (bandicoot, basalt, blesmol)	One down (bandicoot, basalt)
16	10	(20, 9, 18)	(40, 40)
16	30	(60, 29, 57)	(120, 120)
50	10	(20, 8, 16)	(40, 40)
50	30	(60, 29, 57)	(120, 120)

One thing we observed was that with split horizon we get better convergence time than without in case of one node failure(120 - With split horizon 150 - without split horizon). There is no visible difference in convergence time in case intitialization. Similar trend is observed when we vary the period from 30 seconds to 10 seconds.

7.2 Test Case 2 - Cyclic graph with single failure

In this test case, four nodes were executed in a cyclic topology as depicted in Figure 5. Again, Table 3 shows the convergence time without using split horizon, while Table 4 shows the results with split horizon. Times without unit are in seconds. Times in microsseconds are followed by μs symbol.

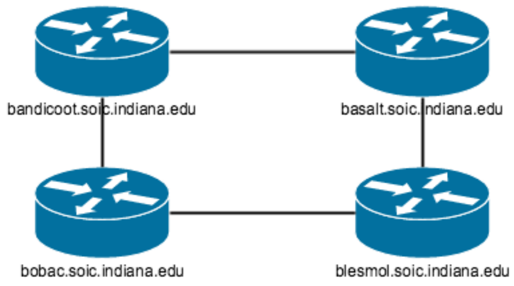

Figure 5: Test Case 2 four nodes cyclic topology

Table 3: Convergence time without split horizon

Infinity	Period	Initialization (bandicoot, basalt, blesmol, bobac)	One down (bandicoot, basalt, bobac)
16	10	(20, 10, 19, 9)	(338μs, 30, 30)
16	30	(60, 28, 56, 25)	(1947μs, 387μs, 390μs)
50	10	(20, 9, 18, 7)	(616μs, 30, 30)
50	30	(60, 28, 56, 24)	(3528μs, 392μs, 390μs)

Table 4: Convergence time with split horizon

Infinity	Period	Initialization (bandicoot, basalt, blesmol, bobac)	One down (bandicoot, basalt, bobac)
16	10	(20, 8, 17, 7)	(30, 30, 40)
16	30	(60, 28, 57, 26)	(90, 90, 120)
50	10	(10, 9, 8, 8)	(20, 20, 41)
50	30	(30, 27, 25, 24)	(60, 60, 120)

No in this test case a different trend is observed as this is cyclic with graph. The convergence time in case of without split horizon is way much faster than that with split horizon attributed to triggred updates. This may look like positive trend but its not cause even though it converges faster but while doing that it creates routing loops which can be dangerous in real time environment. Split horizon even though converging slower is stable at each point i.e very less instances of loops which fade away slowly. This trend is also the same with reduced period of 10 seconds.

7.3 Test Case 3 - Multicyclic graph with multiple failure

In this test case, five nodes were executed in a multicyclic topology as depicted in Figure 6. Table 3 shows the convergence time without using split horizon, while Table 4 shows the results with split horizon. Again, times without unit are in seconds while times in microsseconds are followed by μs symbol.

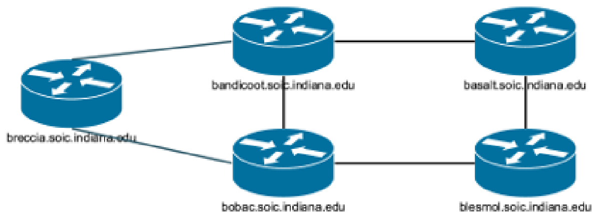

Figure 6: Test Case 3 five node multicyclic topology

Table 5: Convergence time without split horizon

Infinity	Period	Initialization (bandicoot, basalt, blesmol, bobac, breccia)	One down (basalt, blesmol, bobac, breccia)	Two down (basalt, blesmol, bobac)
16	10	(10, 9, 19, 8, 7)	(246μs, 664μs, 267μs, 758μs)	(10, 1647μs, 10)
16	30	(30, 29, 59, 28, 27)	(197μs, 1297μs, 266μs, 1898μs)	(485μs, 270μs, 469μs)
50	10	(10, 9, 19, 8, 7)	(224μs, 659μs, 341μs, 949μs)	(1304μs, 851μs, 1310μs)
50	30	(30, 29, 59, 28, 27)	(234μs, 699μs, 273μs, 818μs)	(395μs, 8208μs, 400μs)

Table 6: Convergence time with split horizon

Infinity	Period	Initialization (bandicoot, basalt, blesmol, bobac, breccia)	One down (basalt, blesmol, bobac, breccia)	Two down (basalt, blesmol, bobac)
16	10	(10, 10, 19, 30, 7)	(180, 160, 130, 170)	(230, 230, 230)
16	30	(30, 29, 58, 28, 27)	(300, 300, 210, 270)	(301, 301, 301)
50	10	(10, 10, 10, 20, 10)	(130, 130, 100, 140)	(485, 270, 469)
50	30	(30, 29, 58, 28, 27)	(300, 240, 515, 270)	(420, 420, 420)

With this Test case we do multiple node failures. As this topology comprises of graphs and many nodes again without split horizon the convergence is faster due to triggred updates but it does create routing loops along the way. The convergence time is more or less the same in case one node failure and two node failure. This trend is also the same with reduced period of 10 seconds.