# Documentation

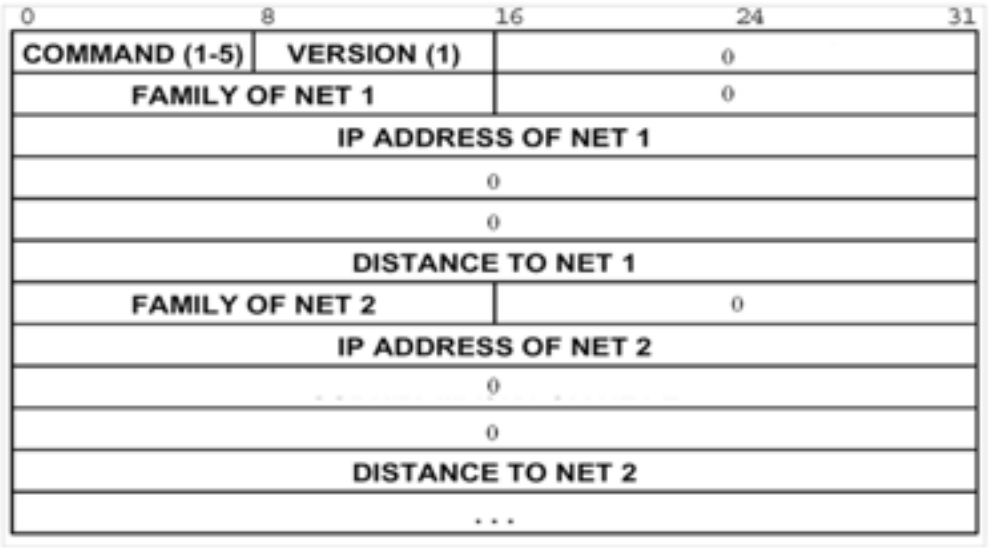
1. A hard copy of all your program files. **done**
2. Emailed tar.gz/zip of all of your code. **done**
3. Include the output for at least one 5-node network with a dump of the routing tables. **done in hard copy.**
4. Demonstrate that the routing converges when a node is brought down and up again. **done in hard copy**
5. A hard copy of the output of your program generated by the config file I provide. **done in hard copy**

**6. Design decisions and implementation:**

**Packet formats.**

In the RFC, the RIP 1.0 packet is defined as follow:

**Simplified RIP Version 1 packet format:**




In my implementation, I made a decision to change the packet format. The reason is in this project, we just simulate part of whole RIP functionality, so we don't necessarily need every bits from original definition. My packet is conceptually identical with the original definition, and it contains more information I need, gets rid of information I don't need right now. It is very easy to modify my original code to comply the spec. I just try to keep my implementation conceptually clean so that I can focus on learning RIP protocol and UDP transportation protocol.

```
struct rmsgEntry{
struct sockaddr_in destination;
int metric;
};

struct rmsg{
int arraySize;
```

```
    struct rmsgEntry array[MAX_RTABLE];
    };
```

struct rmsg is my message structure. The arraySize identify how many message entries in the array, struct rmsgEntry array[MAX_RTABLE]. MAX_RTABLE is #define in .h file which represent the size of routing table I could have. In this implementation is 10.

Each message entry contains information of destination and its metric. I choose to encode destination address into struct sockaddr_in format because in the future for real routing, I can just use this information to send UDP packet right away.

**Describe your design decisions and the consequent limitations. For example, did you use linked-lists or arrays, or what is the maximum number of neighbors that you can handle, or are the values of parameters (e.g., infinity) in the code configurable.**

**Array:**

In the project 1, I tried to used linked list for dynamic creation and deletion data structure so that I faced many memory issues. This time, I choose to use array. So every table in the implementation will be created as an array by pre-defined metric in .h file. This decision put a limitation about following aspects: (arrange from lower,detail to higher level)

- How many char I can take in a getline call?
- How many entries I could have for virtual table?
- How many neighbors I could have?
- How many nodes I can handle in a topology to form a routing table?

These pre-defined values will need manual configurations when deploying to different topology and different input file.

**Timer:**

First, in RFC it talks about 3 timer: update timer, timeout for route, garbage collection for route.

- update timer == 30 s
- timeout == 180 s
- gc = 120 s

In my implementation, I want to facilitate the processing temple, so I decrease each timer by factor of 10. So they become:

- update timer == 3 s
- timeout == 18 s
- gc = 12 s

By doing so, I don't have to wait so long to see convergence.

According to RFC, I need a timer for sending out RIP message per node. And for every entries in the routing table, I need 1 timeout and 1 for garbage collection.

I have interesting experiences for finding the right implementation of timer. There are several options.

1. use alarm() and sigaction() system call, asking OS to help keep track on time;

2. use select() timeout
3. use time() function for static time assignment, and periodically check on it.

I choose 3 to do the job.

First I don't want to create multithreads just for counting time. The option 1 sounds very professional, but to me it is an overkill for this project, and more, one process could only create only one alarm at a time. This is the deal breaker because I need two timer per route in the routing table at least.

Option 2 sounds good at the first glance. Non-blocking, set your own time and you can do it as many times as possible. But the truth is, select() is only non-blocking for recvfrom and sendto. If I use it as timer only, it will block everything. Basically, use select as a timer is equivalent to use the system call sleep(). Deal breaking….

Option 3 is the easiest option but the most effective one, and it fits my requirement well. I combine this and select() in my implementation.
Every timer is just a:

```
time_t timer_name
```

When the timer is created and set, the system time will be assigned to the timer. I will periodically check the timer, see if the time difference between the moment I check the timer and the moment this timer is being set is greater than the a number, like UPDATE for 3 sec and so on.

If the timer expired, I will execute different code accordingly. For example, if update timer is expired, I will create and send out new RIP message. If timeout timer is expired, I will make the route as **invalid** but not delete from the routing table. If the garbage collection timer is expired, I will check if the route is **invalid**, if so, delete it.

**Describe how you handle the "counting to infinity" problem.**

According to RFC:

"If A thinks it can get to D via C, its messages to C should indicate that D is unreachable. If the route through C is real, then C either has a direct connection to D, or a connection through some other gateway. C's route can't possibly go back to A, since that forms a loop.By telling C that D is unreachable, A simply guards against the possibility that C might get confused and believe that there is a route through A."

So I implement it in the code. If A is making RIP message for B, I will make the metric for destination whose gateway is B as infinity. By doing so, try to reduce the chance for abnormal metric counting.