DCCRIP: Protocolo de Roteamento por Vetor de Distância

André Fernandes La Rocca Teixeira (2014004107) e Jonatas Cavalcante (2014004301)

1 - Introdução

Este trabalho implementa um roteador chamado DCCRIP, que utiliza roteamento por vetor de distância. O DCCRIP tem suporte a pesos nos enlaces, balanceamento de carga, medição de rotas e reroteamento imediato.

2 - Discussão Técnica

2.1 - Estruturas

2.1.1 - Vizinhos (*neighbors*)

Tabela que contém todos os vizinhos do roteador corrente. Essa tabela consiste de um dicionário, em que a chave são os IPs de cada um dos vizinhos do roteador corrente. Associado a essa chave, tem-se o valor do peso do enlace entre o roteador corrente e o vizinho.

2.1.2 - Tabela de Roteamento (*routing_table*)

Tabela que consiste de um dicionário, em que a chave é o IP de cada um dos outros roteadores e a estrutura associada à chave é um dicionário de custos. Esse dicionário possui como chave o custo de cada uma das rotas conhecidas para o roteador corrente e a estrutura associada à chave é uma lista de elementos *Route* (como mostrado na imagem abaixo). Cada um desses elementos possui os seguintes dados: custo, próximo salto, tempo de vida e uma *flag* enviado. Para melhor compreensão, considere o seguinte exemplo:

```
{
   '127.0.1.1': {
      10: [
         (10, '127.0.1.1', π, False),
         (10, '127.0.1.2', π, False)
      ],
      20: [
         (20, '127.0.1.1', π, False)
      ]
}
```

```
class Route:
    cost = 0
    nextHop = ""
    time_to_live = 0
    sent = False

    def set(self, cost, nextHop):
        self.cost = cost
        self.nextHop = nextHop
        self.time_to_live = PERIOD
        self.sent = False
```

2.2 - Atualizações Periódicas

No início do programa, a função *send_update_message* é chamada e, através de uma *thread*, ela é chamada novamente a cada período de tempo, que é o valor passado por parâmetro na chamada do programa. Essa função percorre a tabela *neighbors* e realiza a chamada da função *send_message*, que utilizando a melhor rota conhecida, envia a mensagem para o vizinho corrente.

```
def send_update_msg():
    for key, value in neighbors.iteritems():
        send_message(key, create_update_msg(key))
    threading.Timer(float(PERIOD), send_update_msg, ()).start()
```

2.3 - *Split Horizon*

Para a implementação dessa otimização, a função que cria a mensagem de atualização (*create_update_message*) faz a chamada da função *create_distances_table* que recebe o IP de destino da mensagem de update.

```
def create_update_msg(destination):
    return json.dumps({
        'type': 'update',
        'source': ADDR,
        'destination': destination,
        'distances': create_distances_table(destination)
    })
```

Essa função monta a tabela *distances* que faz parte do corpo da mensagem de *update*. A tabela de roteamento é percorrida e todas as rotas são consultadas. Primeiramente, o laço que percorre todos os IPs conhecidos não percorre o IP do destino da mensagem do *update*, assim sendo evita-se que as mensagens enviadas ao vizinho ***x*** tenha rotas para ***x***.

```
def create_distances_table(destination):
    distances_table = {}
    for ip, costs in routing_table.copy().iteritems():
        if ip != destination:
            found_shortest_distance = False
            keylist = costs.keys()
            keylist.sort()
            for key in keylist:
                routes = costs[key]
                for route in routes:
                    if route.nextHop != destination:
                        distances_table[ip] = route.cost
                        found_shortest_distance = True
                        break
                if found_shortest_distance:
                    break
    return distances_table
```

Para as demais rotas consultadas, caso o campo *nextHop* seja igual ao roteador de destino, a rota não é considerada, dessa forma evita-se incluir rotas aprendidas do roteador ***x***, uma vez que as rotas aprendidas por esse roteador necessariamente precisam ter o campo *nextHop* igual ao IP dele. Como o dicionário de custos é ordenado pelos custos, percorre-se as rotas com menor custo primeiramente. Se uma rota de menor custo é encontrada e ela não é aprendida do roteador ***x***, ela é adicionada à tabela *distances* e o laço avança para o próximo IP.

2.4 - Balanceamento de Carga

Para a implementação desse recurso, faz-se uso da tabela de roteamento (*routing_table*). Quando uma mensagem de *update* vai ser enviada para o roteador ***x***, acessa-se o dicionário com a chave ***x***. Uma vez acessado, os custos são ordenados com o intuito de obter-se a rota com o menor custo. Assim sendo, é acessado a lista das rotas com o menor custo. Varre-se essa lista com o intuito de obter a primeira rota que não foi utilizada, atendo-se para verificar se essa lista possui mais de um elemento. Uma vez encontrada, essa rota é usada e, caso haja mais de uma rota, a *flag* enviado é setada como *true*. Assim sendo, no próximo envio de mensagem de *update*, essa rota não será utilizada, e sim a próxima rota que não tem essa *flag setada*, no caso de mais de uma *flag*. No caso de apenas uma rota, a *flag* não é setada e ela é utilizada novamente, uma vez que não há outra opção.

```
def send_message(address, message):
    costs = routing_table[address]
    destination = ''
    selected_cost = 0
    selected_route = Route()
    keylist = costs.keys()
    keylist.sort()

    for key in keylist:
        selected_cost = key
        routes = costs[key]
        for route in routes:
            if route.sent == False:
                destination = route.nextHop
                if len(routes) > 1:
                    route.sent = True
                selected_route = route
                break
    # Sets all the not selected routes as not sent
    for route in costs[selected_cost]:
        if route != selected_route:
            route.sent = False

    server.sendto(message, (destination, PORT))
```

2.5 - Remoção de Rotas Desatualizadas

Para implementar essa funcionalidade, uma função chamada faz uma varredura na estrutura da tabela de roteamento (*routing_table*) a cada segundo, decrementando o campo de tempo de vida (*time_to_live*) de cada uma das rotas. Caso uma das rotas atinja o tempo igual a 0, que é quando ela fica o período especificado sem receber nenhuma atualização, é chamada a função de remoção de rota.

```
def handle_routing_table():
    for ip, costs in routing_table.copy().iteritems():
        for cost, routes in costs.copy().iteritems():
            for route in routes:
                route.time_to_live -= DEFAULT_TIME
                if route.time_to_live == 0:
                    remove_route(ip, cost, routes, route)
    threading.Timer(float(DEFAULT_TIME), handle_routing_table, ()).start()
```

2.6 - Reroteamento Imediato

Quando um enlace é deletado, a tabela de roteamento é percorrida e, para cada uma das rotas aprendidas pelo roteador vizinho que o enlace está sendo removido, é chamada a função de remoção de rota.

```
def del_link(address):
    for ip, costs in routing_table.copy().iteritems():
        for cost, routes in costs.copy().iteritems():
            for route in routes:
                if route.nextHop == address:
                    remove_route(ip, cost, routes, route)
```

A função de remoção remove a rota passada por parâmetro da lista de rotas. Uma vez removida, é verificado se a lista de rotas com aquele custo está vazia e, caso esteja, aquele par (IP, custo) é removido da estrutura. Após essa remoção, é verificado se há alguma rota com outro custo associado àquele IP. Caso não tenha, significa que aquele IP não tem nenhuma rota possível com o roteador corrente e, portanto, ele é removido da estrutura da tabela de roteamento. Ainda nessa condição, caso o IP esteja na tabela *neighbors*, significa que o vizinho foi desconectado do roteador corrente e portanto a entrada correspondente da tabela é removida. Com essa atualização da estrutura de dados, no próximo envio para determinado IP, o algoritmo fará as buscas de melhores rotas nessa base atualizada e automaticamente utilizará a melhor rota remanescente.

```
def remove_route(ip, cost, routes, route):
    routes.remove(route)
    # Verifies if the routes array is empty
    if not routes:
        del routing_table[ip][cost]
        # Verifies if there is any other route to that ip address
        if len(routing_table[ip]) == 0:
            del routing_table[ip]
            # Verifies if the empty link is in the neighbors table and removes if necessary
            if neighbors.has_key(ip):
                del neighbors[ip]
```

3 - Conclusão

O trabalho se mostrou bastante desafiador e promoveu bastante aprendizado à equipe. Os desafios técnicos de implementação proporcionaram uma visão prática muito interessante, que trouxeram conhecimentos complementares à parte teórica da disciplina.