**Universidade Federal de Minas Gerais**
**EE - Engenharia de Sistemas**
**DCC023 - Redes de Computadores**

# Trabalho Prático II - DCCRIP: Protocolo de Roteamento por Vetor de Distância

Humberto Monteiro Fialho - 2013430811
Rafael Carneiro de Castro - 2013030210

6 de junho de 2018

## 1 Introdução

Neste trabalho será implementado o DCCRIP, um roteador que utiliza o roteamento por vetor de distância. O DCCRIP tem suporte a pesos de enlace, balanceamento de carga, medição de rotas e outras funcionalidades. A linguagem utilizada para o desenvolvimento do trabalho foi o Python.

## 2 Desenvolvimento

Como foi recomendado na especificação do projeto, a biblioteca *JSON* foi utilizadas para facilitar o desenvolvimento do trabalho, no momento de se enviar mensagens. A seguir alguns aspectos do código são apresentados.

### 2.1 Estrutura de Dados

Para a implementação do roteador, uma classe chamada Router foi criada. Ela contem o IP do roteador, porta de operação, período de envio da mensagem de atualização de rotas, tabela de histórico de rotas, tabela de roteamento, tabela de nós vizinhos, e os métodos relevantes à operação. Uma importante decisão tomada para a implementação foi no que diz respeito à manutenção das tabelas de histórico e de roteamento. Como será discutido mais para frente, a tabela de histórico é importante para o Reroteamento Imediato. Contudo o código ficaria muito complexo e ilegível caso as duas tabelas recebessem a manutenção necessária a cada entrada de novas rotas, considerando os diversos casos possíveis das rotas vindo dos nós vizinhos. Sendo assim, optou-se por tornar a tabela de histórico a tabela central de funcionamento do roteador. Ela contém todas as informações de rotas recebidas pelos vizinhos, sendo elas as melhores rotas ou não. Sempre que existe uma mudança de peso de uma rota para dado destino através de um vizinho, esta informação é atualizada na linha correspondente da tabela, de forma que ela vai ter as distâncias atualizadas aos destinos a partir do vizinho correspondente. A tabela de roteamento é então construída sob demanda. Sempre que ela é solicitada por algum método, existe um processamento na tabela de roteamento que vai extrair as melhores rotas para cada destino, como se pode ver a seguir.

```
def update_routing_table(self):
    # get the best options of routes for each IP
    # each one can have more than one route with the same distance (load balancing)
    routes_by_ip = dict()
    for history in self.history:
        ip_key = history['ip']
        if ip_key not in routes_by_ip.keys() or routes_by_ip[ip_key][0]['distance'] > history['distance']:
            # if there isn't a route for that IP on the routing table,
            # or the new history has a better distance, update de routing entry
            routes_by_ip[ip_key] = [history]
        elif routes_by_ip[ip_key][0]['distance'] == history['distance']:
            # if already exists a route for that IP with that distance, add new option
            routes_by_ip[ip_key].append(history)

    self.routing_version = self.history_version
    self.routing = routes_by_ip
```

No código, todas as entradas da tabela de histórico são percorridas. Caso o destino da entrada atual possa ser alcançado por uma distância menor à que já está registrada na tabela de roteamento, a entrada da tabela de roteamento é substituída por esta. Caso ainda não exista uma rota para aquele destino registrada na tabela de roteamento, a atual é registrda. Caso a entrada atual possua mesma distância

das já registradas na tabela de roteamento, ela é adicionada como mais uma opção de rota com aquela distância ótima, comportamento importante para o Balanceamento de Cargas, como será discutido mais a frente. Para facilitar a lógica deste processamento, optou-se por implementar a tabela de roteamento como um dicionário, onde a chave é o nó de destino, e o valor é um vetor das opções possível com mesma distância para este destino. Esta decisão também torna mais simples a escolha de uma rota para um destino.

O inconveniente desta abordagem de atualização sob demanda é que o processamento de construção da tabela de roteamento a partir do histórico aconteceria sempre que se precisasse consultar a tabela de roteamento. Como medida paleativa, tanto a tabela de histórico quanto a de roteamente têm um inteiro de controle de versão. Assim, sempre que a tabela de roteamento for solicitada, a versão dela é comparada à versão do histórico. Caso sejam iguais, retorna-se apenas a tabela de roteamento que está em memória. Caso a versão da tabela de roteamento seja menor do que a da tabela de histórico, é preciso atualizá-la, com o resultado da atualização sendo guardado em memória. Para garantir tal comportamento, a tabela de roteamento apenas deve ser acessada pelo método get_routing_table para que tais comportamentos sejam garantidos, como se pode ver no código a seguir. Sempre que um novo registro é adicionado, modificado ou removido da tabela de histórico, sua versão deve ser somada.

```
def get_routing_table(self):
    # updating routing on-demand
    if self.routing_version < self.history_version:
        # update old routing table
        self.update_routing_table()
    return self.routing
```

## 2.2 Atualizações periódicas

Para o correto funcionamento do algoritmo de roteamento, o nó precisa mandar para seus vizinhos periodicamente as distâncias que ele conhece para cada destino, ou seja, mandar a mensagem de update. O período deste envio é passado por parâmetro quando se inicia a aplicação para um nó. Para chamar o método de envio da mensagem de update, um método auxiliar foi criado, como se pode ver a seguir.

```
# method to call function every 'secs' seconds
def set_interval(func, secs):
    def function_wrapper():
        func()
        set_interval(func, secs)
    t = threading.Timer(secs, function_wrapper)
    t.start()
    return t
```

Este método chama a função passada por parâmetro no período também passado por parâmetro, repetidamente. Como este método se utiliza de mecanismos de *thread* do Python, o envio da mensagem de update ocorrerá paralelamente a qualquer outra operação executada pelo código, garantido que seja enviada sempre no tempo especificado. A construção da mensagem de update será vista na sessão *Split Horizon*.

## 2.3 *Split Horizon*

O Split Horizon é uma lógica de contingência que pode ser implementada para se evitar que mensagens de update fiquem circulando na rede em loops infinitos. Para tanto, quando se envia uma mensagem de update para um nó viznho, não se deve enviar a distância conhecida para este próprio nó vizinho, nem as distâncias aprendidas por este nó vizinho, ou seja, as distâncias das rotas que passarão por ele. Para garantir este comportamento, o seguinte método foi implementado na classe Router:

```
def send_update(self):
    routing_table = self.get_routing_table()

    update_message = dict()
    update_message['type'] = 'update'
    update_message['source'] = self.ip
    update_message['destination'] = ''
    update_message['distances'] = dict()

    for ip in routing_table.keys():
        update_message['distances'][ip] = routing_table[ip][0]['distance']

    connection = sck.socket(sck.AF_INET, sck.SOCK_DGRAM)
    for neighbor in self.neighbors:
        # copy message to use original in other neighbor messages
        copy_message = copy.deepcopy(update_message)
        copy_message['destination'] = neighbor['ip']

        # by split horizon, remove route to destination of message
        if neighbor['ip'] in copy_message['distances'].keys():
            del copy_message['distances'][neighbor['ip']]

        # by split horizon, remove route learned from destination of message
        to_remove = []
        for ip in routing_table.keys():
            learned_from_destination = list(filter(lambda option: option['next'] == neighbor['ip'],
                                                   routing_table[ip]))
            if len(learned_from_destination) > 0 and ip in copy_message['distances'].keys():
                to_remove.append(ip)
        for ip in to_remove:
            copy_message['distances'].pop(ip)

        # all routers have the same port
        connection.sendto(json.dumps(copy_message).encode(), (neighbor['ip'], self.port))
```

O método percorre todas as entradas da tabela de roteamento, colocando as distâncias para todos os destinos conhecidos. Para cada vizinho, primeiro se exclui da mensagem a rota para o próprio vizinho, e em seguida se exclui as rotas aprendidas deste vizinho. Por útlimo, a mensagem é enviada. Este método é chamado periodicamente no momento que o router é iniciado, com o auxílio do método set_interval apresentado na sessão anterior.

## 2.4 Balanceamento de carga

Pela estrutura de dados utilizada para a tabela de roteamento, para cada IP de destino tem-se um vetor de opções de rota de mesma distância. Um número aleatório entre 0 e número de opções - 1 é gerado, e a opção correspondente a este número é utilizada para o envio da mensagem. O código com tal lógica pode ser visto a seguir.

```
def send_message(self, message):
    routing_table = self.get_routing_table()

    if message['destination'] in routing_table.keys():
        # select one of the best options for load balancing
        options = routing_table[message['destination']]
        selected_option = randint(0, len(options) - 1)
        selected_hop = options[selected_option]['next']

        connection = sck.socket(sck.AF_INET, sck.SOCK_DGRAM)
        # all routers have the same port
        connection.sendto(json.dumps(message).encode(), (selected_hop, self.port))
```

## 2.5 Reroteamento imediato

Pela estrutura de tabela de roteamento e tabela de histórico criada, o reroteamento se torna bem simples. Basta somar a versão da tabela de histórico quando uma ou mais entradas dela são removidas, o que pode acontecer quando a conexão com nós vizinhos é perdida. Como a versão da tabela de histórico agora é maior, a tabela de rotemanto será atualizada quando for solicitada, conforme visto na sessão Estrutura de Dados.

## 2.6 Remoções de rotas desatualizadas

As rotas desatualizadas são controladas pelo TTL (*time to live*) que cada rota possui. Conforme a especificação, o tempo de vida máximo de uma rota é igual a 4 vezes o período de atualização. Ou seja, quando um roteador recebe uma mensagem de atualização de um vizinho, ele pode primeiramente subtrair o TTL de todas as rotas aprendidas daquele vizinho, e todas as rotas que tem TTL igual a zero são removidas. Em seguida as rotas são atualizadas pela mensagem que chegou do vizinho, e o TTL de todas estas recebidas é colocado em 4, já que pela mensagem recebe-se a confirmação da existência daquela rota. O código de subtração do TTL das rotas aprendidas de um vizinho, junto com a remoção das rotas com TTL igual a zero pode ser visto a seguir.

```
def subtract_ttl(self, source_ip):
    to_remove = []
    # subtract TTL from routes learned from source
    for route in self.history:
        if route['next'] == source_ip:
            route['ttl'] = route['ttl'] - 1
            if route['ttl'] == 0:
                to_remove.append(route)

    # remove routes with TTL equals to 0
    for zero_ttl in to_remove:
        self.history.remove(zero_ttl)
```

O código apresentado na próxima imagem é o código que processa uma mensagem de update. Como se pode ver, ele chama o método subtract_ttl para o controle do tempo de vida das rotas do nó que enviou aquela mensagem de update. Também atualiza a versão do histórico, conforme mencionado na sessão Estrutura de Dados, controlando o valor máximo da versão do histórico.

```
def receive_table_info(self, table_info):
    # find neighbor who sent that information
    source = list(filter(lambda neighbor: neighbor['ip'] == table_info['source'], self.neighbors))
    if len(source) == 0 or table_info['destination'] != self.ip:
        # leave if it's from unknown source or another destination
        return
    source = source[0]

    self.subtract_ttl(source['ip'])

    for ip in table_info['distances'].keys():
        # update the history with the route for that IP by that source
        on_history = list(filter(lambda route: route['ip'] == ip and route['next'] == table_info['source'],
                                 self.history))
        # there should only exists one history for that IP by that source
        if len(on_history) > 0:
            # there is a history for that IP by that source, just update distance and TTL
            on_history[0]['distance'] = table_info['distances'][ip] + source['weight']
            on_history[0]['ttl'] = 4
        else:
            # there isn't a history for that IP by that source, add new
            new_history = dict()
            new_history['ip'] = ip
            new_history['distance'] = table_info['distances'][ip] + source['weight']
            new_history['next'] = table_info['source']
            new_history['ttl'] = 4
            self.history.append(new_history)

    # adding history version to update routing on-demand
    self.history_version = self.history_version + 1
    if self.history_version > MAX_HISTORY_VERSION:
        self.history_version = 0
        self.update_routing_table()
```

# 3 Conclusão

Através deste trabalho foi possível colocar em prática os conceitos do algoritmo de roteamento por vetor de distância. Pelos testes realizados disponibilizados pelo professor, além de casos de teste extras, conclui-se que o resultado final foi satisfatório, e o objetivo do trabalho foi atingido.