Pesquisa e Compressão de Tabelas de Encaminhamento

Gonçalo Ribeiro e Ricardo Amendoeira

28 de Março de 2015

1 Pesquisa da Tabela de Encaminhamento

À data de escrita deste relatório existem cerca de 520K prefixos nas tabelas BGP (Border Gateway Protocol) da Internet. Como tal é essencial que os algoritmos de encaminhamento implementados pelos routers sejam muito eficientes para que os datagramas possam ser encaminhados em tempo útil. Nesta parte do relatório damos conta de uma implementação que fizemos de dois algoritmos de procura em tabelas de expedição (FIB).

O primeiro algoritmo que implementámos organiza os prefixos numa trie binária em que todos os nós excepto as folhas têm sempre dois filhos, uma 2-trie. O prefixo é codificado no caminho desde a raiz até uma folha e cada folha tem o próximo salto correspondente ao prefixo. Desta forma o algoritmo de pesquisa é simples (Algoritmo **??**).

```
node = root
foreach bit in address, starting at MSB do
    if node is not a leaf then
        if bit == 0 then
            node = node.left_child
        else
            node = node.right_child
    else
        return node.interface
end
```

Algoritmo 1: pesquisa de um endereço numa 2-trie

O segundo algoritmo implementa uma trie com compressão de nível (LC-trie) e caminho (Patricia tree). O objectivo é criar uma trie em que as zonas densas — i níveis completos consecutivos — são substituídas por um nó com grau 2^i; e em que as zonas esparsas são comprimidas removendo nós com apenas um filho. Isto resulta numa árvore menos alta e portanto em pesquisas mais rápidas. Esta implementação teve por base [**?**].

Na implementação deste algoritmo é necessário ter em memória a FIB ordenada por prefixo e uma representação da trie. A trie é implementada como um vector e cada nó tem três campos: branch, skip e pointer. Branch refere-se ao grau do nó; skip ao número de nós suprimidos pela compressão de caminho;

pointer à posição no vector do filho mais à esquerda ou, em folhas, à posição da FIB em que pode ser encontrado o next-hop.

A parte mais pesada do algoritmo é a construção da trie. Para calcular o branch de cada nó é preciso varrer os k sufixos do nó actual. Por outro lado o skip de cada nó é $O(1)$ porque basta comparar o primeiro e o último dos k sufixos. Para saber mais detalhes sobre a construção da trie recomendamos a consulta da nossa implementação ou [**?**].

O correspondente algoritmo de pesquisa na trie é o Algoritmo **??**. A função extract(a, p, n) retorna n bits de a começando no bit p, sendo que p=0 se refere ao bit mais significativo de a. Note-se que devido à perda de informação introduzida pela compressão de caminho é preciso sempre verificar na FIB se a folha encontrada corresponde a um prefixo compatível com o endereço que está a ser procurado. Segundo [**?**] para uma FIB com n entradas a altura da LC-trie cresce com $O(log\, log\, n)$ enquanto que numa árvore sem compressão cresce com $O(log\, n)$. Portanto este algoritmo permite pesquisas mais rápidas.

```
node = trie[0]
skip = node.skip
branch = node.branch
pointer = node.pointer
while node is not a leaf do
    node = trie[addr + extract(address, skip, branch)]
    skip = skip + branch + node.skip
    branch = node.branch
    pointer = node.pointer
end
if address starts with fib[pointer].prefix then
    return fib[pointer].nexthop
else
    return DISCARD_PACKET
end
```

Algoritmo 2: pesquisa de um endereço numa LC-trie

2 Compressão da Tabela de Encaminhamento

Para permitir pesquisas mais rápidas implementámos também um algoritmo de compressão de tabelas de encaminhamento, o Optimal Routing Table Constructor (ORTC) desenvolvido pela Microsoft [**?**]. Este algoritmo produz tabelas de encaminhamento óptimas no sentido de não ser possível comprimi-las mais (ver demonstração em [**?**]) e permite em média diminuir em 40% o tamanho de uma tabela. O ORTC comprime as tabelas eliminando redundância e trocando o next-hop geral pelo next-hop mais comum, fazendo depois as alterações necessárias para manter o funcionamento da tabela. O algoritmo tem três passos:

1. Percorrer a árvore da raiz até às folhas transformando-a numa 2-trie, para isso propagando next-hops para os filhos quando necessário. Os next-hops de nós intermédios passam a ser redundantes e podem ser descartados;

2. Percorrer a árvore das folhas até à raiz para calcular qual o next-hop mais utilizado. Cada nó calcula o next-hop mais comum da sua própria sub-árvore uma vez que se explora a árvore em pós-ordem;

3. Percorrer a árvore da raiz até às folhas eliminando as folhas com informação redundante (folhas com next-hops já indicados pelos seus antepassados).

O ORTC tem complexidade $O(n)$ uma vez que a árvore é percorrida três vezes, uma em cada passo. Providenciamos pseudo-código para cada passo nos Algoritmos **??**, **??** e **??**.

```
foreach node N, from root to leaves do
    if N has exactly one child node then
        create missing child node
    if Next-hops(N) == 0 then
        Next-hops(N) = Inherited(N)
end
```

Algoritmo 3: passo 1 do algoritmo ORTC

```
foreach node N, from leaves to root do
    if N's children have a next-hop in common then
        Next-hops(N) =
            Common(Next-hops(N->left),Next-hops(N->right))
    else
        Next-hops(N) =
            Next-hops(N->left) joined with Next-hops(N->left)
end
```

Algoritmo 4: passo 2 do algoritmo ORTC

```
foreach node N, from root to leaves do
    if N is not the root and Inherited(N) is in Next-hops(N) then
        Next-hops(N) = empty-set
    else
        Next-hops(N) = First(Next-hops(N))
end
```

Algoritmo 5: passo 3 do algoritmo ORTC