高效IP路由查找实验

中国科学院大学
袁欣怡 2018K8009929021
2021.5.20

实验内容

基于 forwarding-table.txt 数据集(Network, Prefix Length, Port)，实现高效 IP 路由查找算法，具体包括：

1. 实现最基本的前缀树查找；
2. 调研并实现某种 IP 前缀查找方案；
3. 比较后者在占用内存和运算时间上的提升。

实验流程

1. 搭建实验环境

本实验中涉及到的文件主要有：

- code
    - simple # 实现最基本的前缀树查找
        - forwarding-table.txt # 数据集
        - lookup.c # 最基本的前缀树查找
        - Makefile # 处理make all和make clean命令
        - run.sh # shell脚本，运行可执行文件10次
    - optimize # 实现改进方法Leaf Pushing
        - forwarding-table.txt # 数据集
        - lookup.c # 实现改进方法Leaf Pushing
        - Makefile # 处理make all和make clean命令
        - run.sh # shell脚本，运行可执行文件10次

2. 实验代码设计

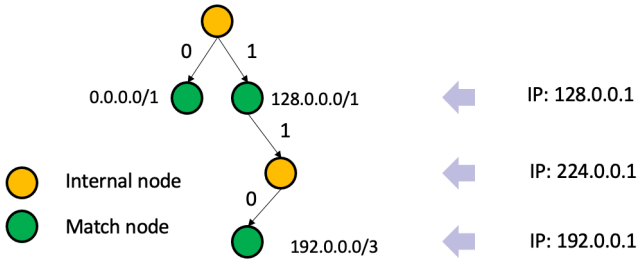
基础设计

路由器在进行路由转发时，需要查找路由表来确定是否有匹配的条目。在之前的实验中，我们已经实现了基础的 IP 查找方法。但是随着网络规模的扩大，线性查找的时间和空间开销都迅速增加，导致效率降低。因此，我们需要减少查找次数，实现更高效的查找算法。

我们这里实现的 1-bit 前缀树有以下特点：

1. 根结点不对应任何 IP，除根结点外每个结点对应一个 IP；
2. 从某结点到它的子结点的路径对应一个字符，从根结点到某一结点的路径上所有字符连接起来，就是该结点对应的 IP；
3. 每个结点对应的 IP 各不相同。

构造出的前缀树如下图所示：



结点数据结构：

```
struct TreeNode{
    int net_id;
    int prefix_len;
    int port;
    struct TreeNode *lchild;
    struct TreeNode *rchild;
    struct TreeNode *parent;
};
```

相应的，为路由表项也设计了数据结构：

```
struct RouterEntry{
    int net_id;
    int prefix_len;
    int port;
};
```

在我们的实验中，需要先扫描 forwarding-table.txt 中的路由表项，构造前缀树，然后进行查找，并计算所需要的时间和空间。代码如下：

```
int main() {
    FILE * fd = fopen("forwarding-table.txt","r"); //读文件

    if (fd==NULL){
        printf("ERROR: File does not exist!");
        return 0;
    }

        // 建立前缀树，计算需要的空间
    char * line = (char *) malloc(MaxLine);

    struct TreeNode * root=NULL;
    root = init_tree();

```

```
    while(fgets(line, MaxLine, fd)!=NULL){
        struct RouterEntry * router=NULL;
        router = line_parser(line);
        add_node(router, root);
    }

    printf("INFORMATION: Memory used: %ld B =%ld KB =%ld
MB\n",mem_use,mem_use/1024,mem_use/1024/1024);

        // 查找前缀树，计算需要的时间
    double time_use;
    double handle_time;
    double avg_time;
    struct timeval start, end;
    int input_num=0;

    //calculate time needed for processing input
    fseek(fd, 0, SEEK_SET);
    gettimeofday(&start, NULL);
    while(fgets(line,MaxLine,fd) != NULL){
        char * net_id=NULL;
        net_id = strtok(line, " ");
        char_to_int(net_id);
        input_num++;
    }
    gettimeofday(&end, NULL);
    time_use = (end.tv_sec-start.tv_sec)*1E9+(end.tv_usec-start.tv_usec)*1E3;
    handle_time = time_use / input_num;

    //calculate time needed in total
    input_num=0;
    fseek(fd, 0, SEEK_SET);
    gettimeofday(&start, NULL);
    while(fgets(line,MaxLine,fd) != NULL){
        char * net_id=NULL;
        net_id = strtok(line, " ");
        lookup(char_to_int(net_id), root);
        input_num++;
    }
    gettimeofday(&end, NULL);
    time_use = (end.tv_sec-start.tv_sec)*1E9+(end.tv_usec-start.tv_usec)*1E3;
    avg_time = time_use / input_num;

    printf("INFORMATION: Average time needed: %.9f ns\n", avg_time-handle_time);

    return 0;

}
```

下面来看 1-bit 前缀树的建立和查找。

（1） 1-bit 前缀树的建立

先初始化前缀树的根结点 root ，然后从数据集中读取表项。从高位开始，如果为 0 则访问左子结点（若不存在则新建一个左子结点），如果为 1 则访问右子结点（若不存在则新建一个右子结点），以此类推，直到访问的长度达到掩码为止。这个过程中同时需要注意设置端口号。

具体代码实现如下：

```
int add_node(struct RouterEntry * router, struct TreeNode * root){
    struct TreeNode * ptr = root;
    int mask = 0x80000000;
    unsigned int prefix_bit = 0x80000000;
    int insert_num = 0;
    int i=0;

    for(i=1; i<=router->prefix_len; i++){
        if ( (router->net_id & prefix_bit)==0 ){
            if (ptr->lchild == NULL){
                ptr->lchild = insert_tree(router, i, ptr);
                insert_num++;
            }
            ptr = ptr->lchild;
        }
        else{
            if (ptr->rchild == NULL){
                ptr->rchild = insert_tree(router, i, ptr);
                insert_num++;
            }
            ptr = ptr->rchild;
        }

        mask = mask>>1;
        prefix_bit = prefix_bit>>1;
    }

    if (insert_num==0)
        printf("net:%x update -> port: %d\n ", router->net_id&mask, ptr->port);

    if ((router->net_id&mask) != (ptr->net_id&mask))
        return -1;

    free(router);
    mem_use -= sizeof(struct RouterEntry);

    return ptr->port;
}
```

（2） 1-bit 前缀树的查找

从根结点出发，与 net_id 的最高位开始匹配。如果最高位为 0 则访问根结点的左子结点，否则访问右子结点，以此类推，最后返回端口号。需要注意的是，如果端口号为 -1，说明查询路径最终落在中间结点上，则要调用 look_back 函数进行回溯。

```
int lookup(int net_id, struct TreeNode *root){
    struct TreeNode * ptr=root;
    unsigned int prefix_bit = 0x80000000;
    int i;
    int port;

    for (i=1; i<32; i++){
        if ((net_id & prefix_bit) == 0){
            if (ptr->lchild == NULL){
                if ((net_id&get_mask(ptr->prefix_len)) != (ptr->net_id&get_mask(ptr->prefix_len))){
                    port = -1;
                    break;
                }
                port = ptr->port;
                break;
            }
            ptr = ptr->lchild;
        }
        else{
            if (ptr->rchild == NULL) {
                if ((net_id&get_mask(ptr->prefix_len)) != (ptr->net_id&get_mask(ptr->prefix_len))){
                    port = -1;
                    break;
                }
                port = ptr->port;
                break;
            }
            ptr = ptr->rchild;
        }
        prefix_bit = prefix_bit >> 1;
    }

    if ((net_id&get_mask(i)) != (ptr->net_id&get_mask(i)))
        port = -1;

    if(port == -1)
        port = look_back(ptr);

    return port;
}
```

look_back 函数的功能是通过回溯找到最近的最长前缀匹配的网络号，并返回该网络对应的转发端口。具体代码如下：

```
int look_back(struct TreeNode * node){
    struct TreeNode * ptr = node;

    while (ptr->port == -1)
        ptr = ptr->parent;

    return ptr->port;
}
```

基于Leaf Pushing的优化

基础的查找函数中， TreeNode 结构中需要存储 prefix_len 、 net_id 和 parent ，但这三项其实都可以删除，这样可以简化 TreeNode 结构，减少占用的内存空间。

新设计的TreeNode结构：

```
struct TreeNode{
    int port;
    struct TreeNode *lchild;
    struct TreeNode *rchild;
};
```

针对这个新的结构，设计的算法也需要进行修改。具体如下：

add_node 函数和 insert_tree 函数：

```
int add_node(struct RouterEntry * router, struct TreeNode * root){
    struct TreeNode * ptr = root;
    // int mask = 0x80000000;
    unsigned int prefix_bit = 0x80000000;
    int insert_num = 0;
    int i=0;

    for(i=1; i<=router->prefix_len; i++){
        if ( (router->net_id & prefix_bit)==0 ){
            if (ptr->lchild == NULL){
                ptr->lchild = insert_tree(router, i, ptr);
                insert_num++;
            }
            ptr = ptr->lchild;
        }
        else{
            if (ptr->rchild == NULL){
                ptr->rchild = insert_tree(router, i, ptr);
                insert_num++;
            }
            ptr = ptr->rchild;
        }

        // mask = mask>>1;
```

```
        prefix_bit = prefix_bit>>1;
    }

    /*
    if (insert_num==0)
        printf("net:%x update -> port: %d\n ", router->net_id&mask, ptr->port);

    if ((router->net_id&mask) != (ptr->net_id&mask))
        return -1;
    */

    free(router);
    mem_use -= sizeof(struct RouterEntry);

    return ptr->port;
}

// 新的insert_tree函数只保存端口号。倘若某个结点是中间结点，则只需要继承父结点的端口号即可
struct TreeNode * insert_tree(struct RouterEntry * router, int prefix_len, struct TreeNode *parent){
    struct TreeNode * node = (struct TreeNode *) malloc(sizeof(struct TreeNode));
    mem_use += sizeof(struct TreeNode);

    // node->net_id = router->net_id & get_mask(prefix_len);
    // node->prefix_len = prefix_len;
    if (router->prefix_len != prefix_len)
        // node->port = -1;
        node->port = parent->port;
    else
        node ->port = router->port;

    // node->parent = parent;
    return node;
}
```

lookup 函数：删除了回溯查找的部分。因为中间结点保存了最近的最长前缀匹配结果所对应的端口号，所以只要进行简单的前缀查找即可。

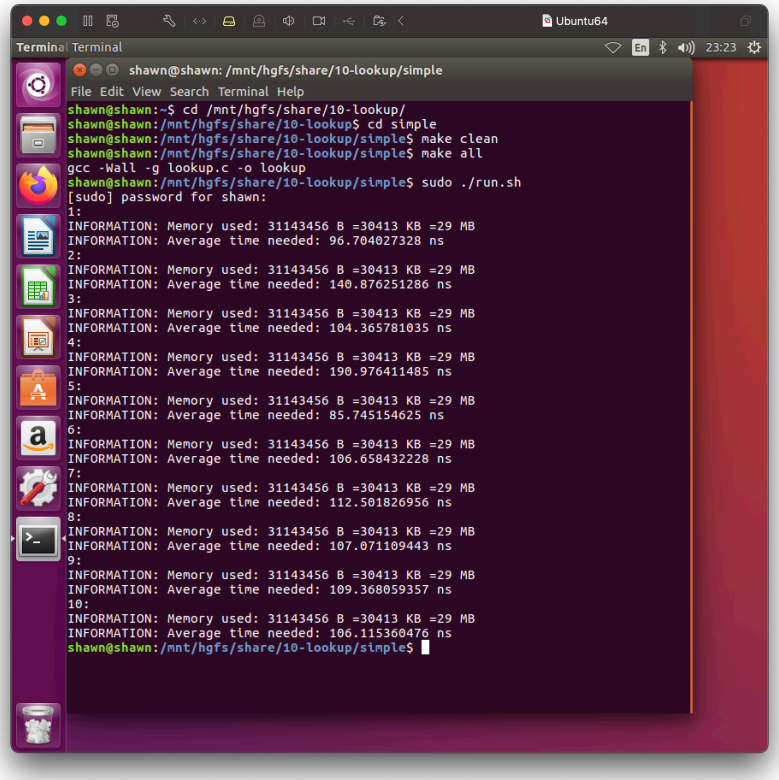
```
int lookup(int net_id, struct TreeNode *root){
    struct TreeNode * ptr=root;
    unsigned int prefix_bit = 0x80000000;
    int i;
    int port;

    for (i=1; i<32; i++){
        if ((net_id & prefix_bit) == 0){
            if (ptr->lchild == NULL){
                /*
                if ((net_id&get_mask(ptr->prefix_len)) != (ptr->net_id&get_mask(ptr->prefix_len))){
                    port = -1;
```

```
13                  break;
14              }
15              */
16              port = ptr->port;
17              break;
18          }
19          ptr = ptr->lchild;
20      }
21      else{
22          if (ptr->rchild == NULL) {
23              /*
24              if ((net_id&get_mask(ptr->prefix_len)) != (ptr->net_id&get_mask(ptr->prefix_len))){
25                  port = -1;
26                  break;
27              }
28              */
29              port = ptr->port;
30              break;
31          }
32          ptr = ptr->rchild;
33      }
34      prefix_bit = prefix_bit >> 1;
35  }
36
37  /*
38  if ((net_id&get_mask(i)) != (ptr->net_id&get_mask(i)))
39      port = -1;
40
41  if(port == -1)
42      port = look_back(ptr);
43  */
44
45  return port;
46 }
```

3. 实验结果分析

基础前缀树查找：

运行 run.sh 脚本，将 lookup 执行十遍，实验结果：

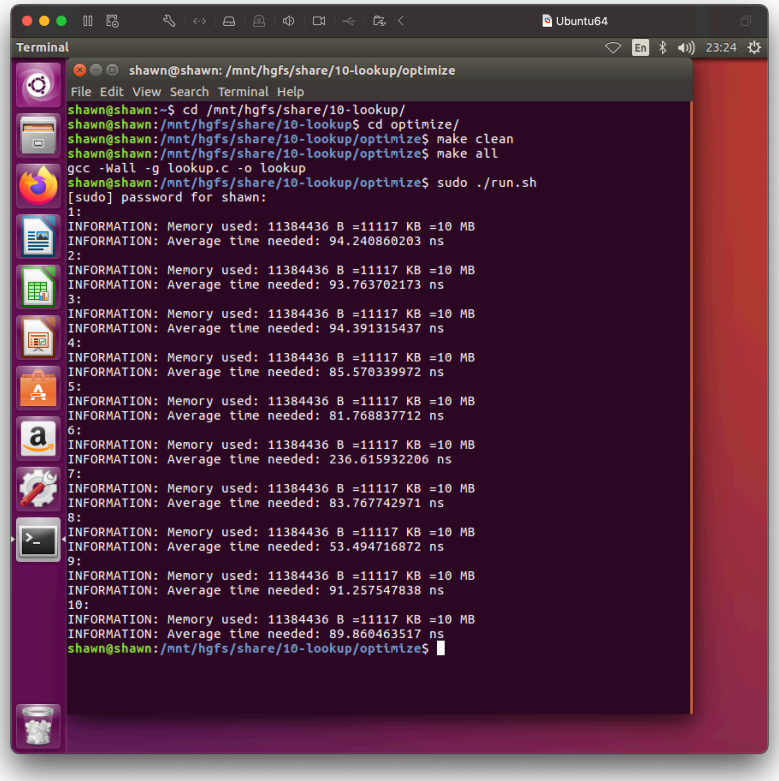


平均内存开销为：31143456B = 30413KB = 29MB

平均花费时间为：116.04ns

基于Leaf Pushing优化

运行 run.sh 脚本，将 lookup 执行十遍，实验结果：



平均内存开销为：11384436B = 11117KB = 10MB

平均花费时间为：100.47ns

比较

平均内存开销减少明显，平均花费时间略有减少。

原因：1. TreeNode 结点内容减少了一半左右，导致内存开销大幅降低；2. 查找过程中减少回溯的次数，但是可以减少的次数不多，因此时间有减少但不太明显。

参考资料

1. 数据结构与算法：字典树（前缀树）： https://zhuanlan.zhihu.com/p/28891541
2. 计算机网络复习——路由器和路由查找算法： https://blog.csdn.net/not_simple_name/article/details/103745833