Computer Networks 94 (2016) 62–79

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

CONSERT: Constructing optimal name-based routing tables

Huichen Dai, Bin Liu*

Department of Computer Science and Technology, Tsinghua University, China

ARTICLE INFO

Article history:
Received 7 April 2015
Revised 15 November 2015
Accepted 18 November 2015
Available online 2 December 2015

Keywords:
Name-based routing table
Optimal
Compressing

ABSTRACT

Name-based routing belongs to a routing category different from address-based routing, it is usually adopted by content-oriented networks [Sharma et al., 2014, Koponen et al., 2007, Rajahalme et al.,2011, Thaler et al.,1998, Hwang et al., 2010, Gritter et al., 2001, Caesar et al., 2006, Carzaniga et al., 2004, Koponen et al., 2007, Hwang et al., 2009 Singla et al., 2010, Detti et al., 2011, Jain et al., 2011 Xu et al., 2013, Katsaros et al., 2012. [1–15]] e.g., the recently proposed Named Data Networking (NDN). It populates routers with *name-based routing tables*, which are composed of *name prefixes* and their corresponding next hop(s). Name-based routing tables are believed to have much larger size than IP routing tables, because of the large amount of name prefixes and the unbounded length of each prefix. This paper presents CONSERT—an algorithm that, given an arbitrary name-based routing table as input, computes a routing table with the *minimal* number of prefixes, while keeping equivalent forwarding behavior. The optimal routing table also supports incremental update. We formulate the CONSERT algorithm and prove its optimality with an induction method. Evaluation results show that, CONSERT can reduce 18% to 45% prefixes in the synthetic routing tables depending on the distribution of the next hops, and meanwhile improve the lookup performance by more than 20%. Prior efforts usually focus on compact data structures and lookup algorithms so as to reduce memory consumption and expedite lookup speed of the routing table, while CONSERT compresses the routing table from another perspective: it removes the inherent "redundancy" in the routing table. Therefore, CONSERT is orthogonal to these prior efforts, thus the combination of CONSERT and a prior compressing method would further optimize the memory consumption and lookup speed of the routing table. E.g., we can first adopt CONSERT to achieve the optimal routing table, and afterwards apply NameFilter [Wang et al., 2013. [16], a two-stage-Bloom-filter method, to that optimal table. This combination diminishes the memory consumption of the routing table data structure by roughly 88%, and increases the lookup throughput by around 17% simultaneously. The joint method outperforms each individual method in terms of memory savings and absolute lookup throughout increase.

1. Introduction

1.1. Background on name-based routing

Routing in current Internet belongs to the category of address based routing. Different from this practice, Name-Based Routing (NBR) has been proposed in the literature [1–15], and is recently re-investigated by the pioneers of Information Centric Networks [17–26], where each piece of content is assigned a unique name. Just as BGP distributes

* Corresponding author.
E-mail addresses: dhc10@mails.tsinghua.edu.cn (H. Dai), liub@tsinghua.edu.cn (B. Liu).

http://dx.doi.org/10.1016/j.comnet.2015.11.020
1389-1286/

address prefix reachability information among autonomous systems, NBR distributes name prefix reachability to routers.

NBR needs a name-based routing table[1] to route packets, where each entry consists of a name prefix and its corresponding next hop. The names are hierarchically structured (the sub-name at each level is called a *component*), and the longest prefix match (LPM) rule still applies to name lookup. As stated in many previous works [16,27–29], a name-based routing table contains much more name prefixes than the IP prefixes in an IP routing table, and name prefixes have much longer lengths than IPv4/v6 addresses. The result is a name-based routing table of huge size, incurring challenges on memory efficiency, routing lookup throughput, route update performance, etc.

1.2. Motivation

A remarkable challenge that a name-based routing table confronts is its large size, in terms of the number of table entries and the length of each entry. The large size can further degrades the name lookup and packet forwarding performance. In particular, this challenge can be described by the following difficulties. First, names are far more complex than IP addresses. As introduced above, names are much longer than IPv4/IPv6 addresses; each name is composed of tens, or even hundreds, of characters. Moreover, unlike fixed-length IP addresses, content names have variable lengths, which further complicates the name lookup process. Second, name-based routing tables could be much larger than today's IP routing tables. Compared with the current IP routing tables with up to 500K IP prefix entries, name-based routing tables could be orders of magnitude larger [29]. Without elaborate compression and implementation techniques, they can even exceed the capacity of today's commodity memory devices. Third, wire speeds have been relentlessly accelerating. Such large table size will definitely hinder the goal of high-speed name lookup, as well as fast routing table updates. Therefore, we spare no efforts in seeking ways to compress the name-based routing table.

1.3. Our work

In this paper we present an algorithm for constructing an *optimal* name-based routing table that has the least possible number of entries, while still providing the same routing information. **More accurately, the optimality is defined as follows: given an arbitrary name-based routing table, the algorithm produces an optimal one that: (1) has the least possible number of prefixes, (2) has the same forwarding behavior as the original routing table.** We call this algorithm CONSERT (Constructing Optimal Name-baSEd Routing Table, read as "CONCERT"). Actually, CONSERT can be viewed as the generalization of ORTC [30], a pioneer work which computes optimal IP routing tables based on the binary *trie* [31] presentation. A binary trie is basically a binary tree where each edge stands for a bit. CONSERT extends from binary trie to multi-way tree—a generalization on routing table optimization problem, but this extension is non-trivial.

[1] For brevity, from now on we may only say "routing table", but the context can distinguish whether it is IP-based or name-based.

CONSERT makes use of the *component trie* structure to represent a name-based routing table, which is a multi-way tree where each edge stands for a component in the name. It is natural that CONSERT adopts the trie for the name-based routing table, because a tree-like structure resembles the structure of a hierarchical and aggregatable name space. The most important difference between a component trie and a binary trie (adopted by ORTC) is, in a binary trie, a node has at most two children nodes, while in a component trie, a node can have unlimited number of children nodes. This difference hinders ORTC from being adapted for name-based routing table compression, so CONSERT is proposed to accommodate the unlimited number of children nodes, and this is where the novelty of this paper stems from.

CONSERT constructs the optimal routing table by three passes over the trie. *Pass One* introduces a special '#' symbol by creating a '#' child node for all the non-leaf nodes in the trie, then pushes the parent's next hop(s) down to that child node. Therefore, the next hop of the '#' node can be viewed as the *default route* of a sub-tree rooted at its parent node. *Pass Two* pushes the most prevalent next hop(s) upwards as high as possible, and *Pass Three* determines the final next hop for each prefix and outputs the optimal routing table. Initially, CONSERT makes two assumptions: (1) the routing table has a default route, and (2) each prefix has a single next hop, but later we remove these two restrictions. Moreover, we also develop an adapted LPM algorithm (from the conventional LPM) to accommodate an '#' symbol in the optimal routing table. We formulate the CONSERT algorithm and prove its optimality using the induction method.

It is worth pointing out that CONSERT aims to remove the "redundancy" inherently in the original routing table, so as to achieve an optimal one with the fewest number of entries (reflected by the number of solid nodes in the trie). Prior efforts often take advantage of fast lookup algorithms and compact data structures, in order to expedite the lookup process and reduce memory consumption, but the redundancy still remains. This is the distinction that our work differs from these efforts. Hence, CONSERT is orthogonal to the prior compressing algorithms and they can be applied jointly. Specifically, some existing compressing methods, e.g., Bloom filter based ones, have good performance on memory consumption and lookup speed, but cannot handle routing updates. It's appealing that CONSERT deals with updates and produces an optimal routing table at first (on the control plane), afterwards another compressing method is further applied to that optimal table (on the data plane). As we shall see, such cooperation can remarkably promote both compressing and lookup performance.

Specifically, we make the following contributions:

(1) Introduce a special symbol '#' into the trie structure and create a node for it, whose next hop stands for the default route of its parent node. This symbol clears the way for optimal routing table compression to make it possible;
(2) We propose the CONSERT algorithm to build an optimal routing table that has the fewest number of prefixes, while keeping the forwarding behavior unchanged.
(3) Adapt the conventional LPM algorithm to accommodate the '#' symbol.

(4) Formulate the CONSERT algorithm and prove its optimality by the induction method.

Experimental results show that CONSERT can reduce roughly 18%–45% prefixes in the synthetic name-based routing tables, and this compression ratio still reaches 30% when prefixes have multiple next hops. Having fewer prefixes reduces the size of the forwarding data structure, especially when CONSERT is applied with certain orthogonal method together. E.g., the combination of CONSERT and NameFilter [16], a two-stage-Bloom-filter method, can jointly reduce the memory consumption of routing tables by around 88%. Meanwhile, since CONSERT reduces the amount of prefixes and shortens the prefix length, the name lookup performance is increased at the same time. Lookup throughput based on the optimal trie output by CONSERT can increase by more than 20%, while the joint method of CONSERT and NameFilter can improve the lookup throughput by around 17% compared with the NameFilter.

The rest of the paper is organized as follows. Section 2 describes the CONSERT algorithm, Section 3 improves the algorithm and Section 4 addresses the routing table updates. Section 5 evaluates CONSERT in terms of memory savings and lookup throughput improvement. Section 6 surveys related work and Section 7 concludes the paper. The proof of the optimality of CONSERT is provided in the Appendix.

2. Construct optimal name-based routing tables

2.1. Preparation – trie of different granularities

Before elaborating on the CONSERT algorithm, we need some preparations on the data structure to organize the name-based routing table. The names, as aforementioned, are hierarchically structured and have unbound lengths. E.g., `org/journal/2016/cfp.html` is a legitimate name, where `org`, `journal`, `2016` and `cfp.html` are 4 *components* of the name. IP routing tables often adopt the binary trie representation, which is basically a binary tree where each edge stands for one bit. Name-based routing table can also take advantage of the trie structure to represent itself, while each trie edge stands for a component in the name, and we refer to this kind of trie as *component trie*. Component trie matches the hierarchical and aggregatable name space of name-based routing tables.

If we give up the hierarchy information in the name prefixes, a name-based routing table can also be represented by a trie of finer granularities – each trie prefix stands for a character or a bit, and we call them *character trie* and binary trie (which we already know), respectively. For brevity we hereby assume that the names in the routing tables consist of only case-insensitive alphabet characters.[2] Each node can have numerous children in the component trie, while at most 26 for the character trie and at most 2 for the binary trie. The CONSERT algorithm mainly targets at the component trie, and we will also adapt it to character trie and binary trie for comparison purpose. The majority of the algorithms are the same, only some minor parts are specific to each kind of trie.

Table 1
Routing Table.

Prefix	Next Hop
/google	4
/google/maps	1
/google/mail	3
/google/scholar	2
/google/news	2
/google/image	2
/yahoo	2
/yahoo/sports/nfl	1
/yahoo/sports/nba	3
/yahoo/sports/nhl	3
/yahoo/sports/mlb	3
/bing	4
/apple	4
/twitter	4
*	5

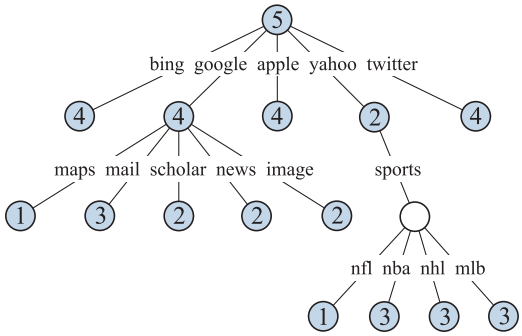


Fig. 1. Component trie representation of the name-based routing table.

The following subsections will describe the CONSERT algorithm step by step in combination of a concrete example. Table 1 shows a typical name-based routing table, and its corresponding component trie representation is illustrated in Fig. 1. We label *prefix nodes* with next hop information (an integer or a set of integers) by *solid* or *occupied* nodes, while the non-prefix nodes are *empty*[3].

In the simplest form, we assume each prefix in the routing table has a single next hop, and the routing table has a default route. These two assumptions will be removed later in Section 3. CONSERT optimizes a name-based routing table using three passes over the trie representation, and below we elaborate on them.

2.2. Pass One

The first pass converts the component trie representation of the routing table into a format suitable for compression. For each *non-leaf* node v in the trie (including the root), it creates a new child node w for a special component '#', and the next hop information for w inherits from v. If node v does not have next hop information, then inherits from v's closest

[2] If the character set expands to include more elements, CONSERT also works.

[3] Note that an empty node means the corresponding prefix does NOT exist in the routing table, but it does have a next hop according to the LPM rule, e.g., `/yahoo/sports` has a next hop of 2.

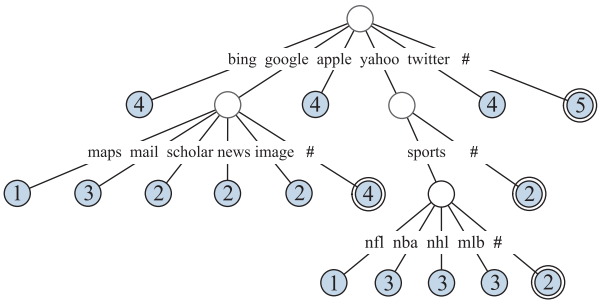


Fig. 2. Component trie representation after Pass One. Newly created nodes are highlighted by double circles.

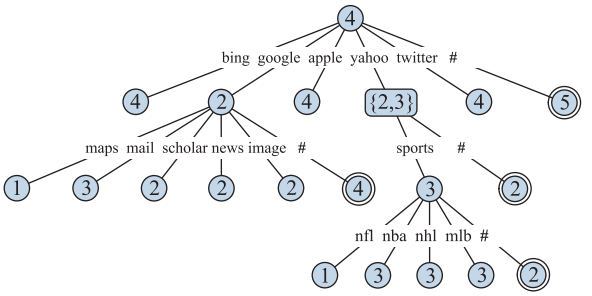


Fig. 3. Component trie representation after Pass Two.

Algorithm 1 Pass One.

```
1: procedure PassOne(Trie trie)
2:     for each non-leaf node v in trie do
3:         create a new child note w for component '#';
4:         if v.next_hop ≠ NULL then
5:             w.next_hop ← v.next_hop;
6:             v.next_hop ← NULL;
7:         else
8:             v ← v's closest solid ancestor;
9:             w.next_hop ← v.next_hop;
```

ancestor node that is not empty. We denote the newly added edge by '#' in the trie, the meaning of this special symbol will be described later. After each '#' child node inherits a next hop, its parent's next hop information is no longer needed and hence discarded. In this way, all the internal nodes plus the root no longer have next hop information. The trie representation after Pass One is illustrated in Fig. 2, where the newly created nodes are highlighted by double circles, and the Pass One algorithm is shown in Algorithm 1.

The routing table after Pass One has exactly the same information as the original one. Due to the introduction of '#', the next hop of a node is "pushed" into the '#' child, so the LPM algorithm will slightly be adapted: any unmatched component (including the *empty* component) will go to the branch of '#', which means the next hop in the '#' node is the default route of the sub-tree rooted at its parent node v. Therefore, at node v, if the next input component is empty[4] (meaning there is no further input component), the next hop of the '#' node should be returned; if the next input component does not match any of v's component (excluding the '#' node), then the next hop of the '#' node should still be returned. According to this lookup rule, we can derive the same next hop for a name on the trie output by Pass One. The final adapted LPM algorithm will be presented in Section 2.5.

[4] This case can happen in name lookup because the input name has variable length, it may equal a prefix represented by an internal node in the trie. But in IP lookup this case cannot happen because the input IP address has fixed length.

Algorithm 2 Pass Two.

```
1: procedure PassTwo(Trie trie)
2:     for each node v in trie (from leaves to root) do
3:         if v is a parent node then
4:             v.next_hop ← v.PrevalentSelect()
```

2.3. Pass Two

The second pass calculates the most *prevalent* next hop among a node's children nodes based on the output trie of Pass One, and take such next hop as that node's candidate next hop. The most prevalent next hop formally means that such next hop has the most occurrences among all the children nodes of a parent node. The algorithm is shown in Algorithm 2. This operation is done at every level from the bottom level up to the tree root, so an easy implementation could be a bottom-up traversal. At each parent node v visited during the traversal, a set of next hops is calculated by the `v.PrevalentSelect()` function, which accomplishes the task of selecting the most prevalent next hop among the children nodes of v.

If `v.PrevalentSelect()` finds that more than one next hops tie for most prevalent among v's children nodes, then they are jointly carried up to the parent node v as candidate next hops. Extremely, if node v's children nodes all have unique next hops, i.e., all the next hops have a population of 1, then they are all carried up to node v, so v has multiple candidate next hops. When v's parent node w invokes `w.PrevalentSelect()`, it will take all the candidate next hops (if more than one) of each child node into consideration and continue to select the most prevalent one(s), so on and so forth. When the second pass is complete, every node in the trie is labeled with a set of candidate next hop(s). Fig. 3 shows the result of the example routing table after Pass Two. Take the node corresponding to `/yahoo/sports` as an example, because 3 out of its 5 children has the next hop 3, 3 is selected as its potential next hop. For the same reason, node corresponding to `/google` selects 2 and the root node selects 4 as their potential next hops, respectively. While for the node corresponding to `/yahoo`, the next hops 2 and 3 from its two children both have a population of 1, so they are equally most prevalent and both carried up as the parent node's candidate next hops.

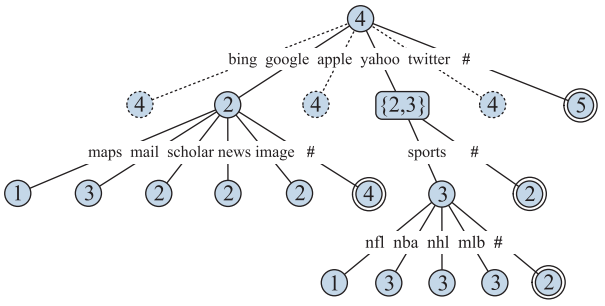


Fig. 4. During Pass Three: selecting 4 as next hop for root. Dotted nodes and edges are to be deleted.

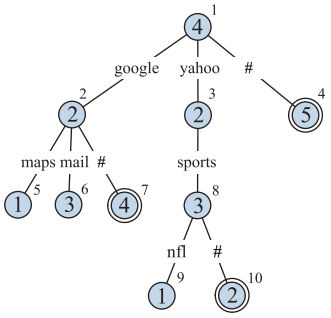


Fig. 5. Optimal routing table after Pass Three.

2.4. Pass Three

The third pass selects final next hops for prefixes and eliminates redundant routes, from the root to the bottom level. This could be realized by a top-down traversal by levels. Each node visited in Pass Three will eventually have a set of final next hops, which is in fact a subset of those computed in Pass Two. Denote by P_v the set of next hops calculated by Pass Two for node v.

The algorithm is as follows. For the root node, we randomly pick up an element from its P_v, and set it as the root's final next hop. For the rest nodes, each node v will inherit a next hop p from the closest non-empty ancestor node. If p is a member of v's set of potential next hops ($p \in P_v$), then node v does not need a next hop by itself. Because a match on this node will inherit the final next hop from its closest non-empty ancestor node, so v will be set to empty. If v is a leaf node, it will be deleted from the trie, as well as its edge. Whenever a node v is deleted (except the '#' node), its parent node w should be notified and node w keeps track of v's corresponding component in a set S_w. Therefore, set S_w records node w's removed children components. Otherwise ($p \notin P_v$), then node v really needs a next hop and it will be set to occupied. We also randomly pick up a member from P_v as the final next hop for node v, just as the root node does. The above process is illustrated in Algorithm 3.

Figs. 4 and 5 illustrate the results during and after Pass Three. (The number on the top right corner of a node in Fig. 5 is the node ID.) After Pass Two, the root node is labeled with a next hop set {4}, so Pass Three can only select the next hop 4 for the root. Fig. 4 elucidates this intermediate result of the trie, where dashed nodes and edges mean that they are to be pruned, and the corresponding components are kept in $S_{root} = \{\texttt{bing, apple, twitter}\}$. After three passes, an optimal routing tables is constructed, as illustrated in Fig. 5. The optimal routing table has 10 entries, while the original one in Table I has 15 entries – 33.33% of the prefixes are reduced. It's so important to point out that according to the adapted lookup algorithm in Algorithm 4, the next hop for each prefix remains the same.

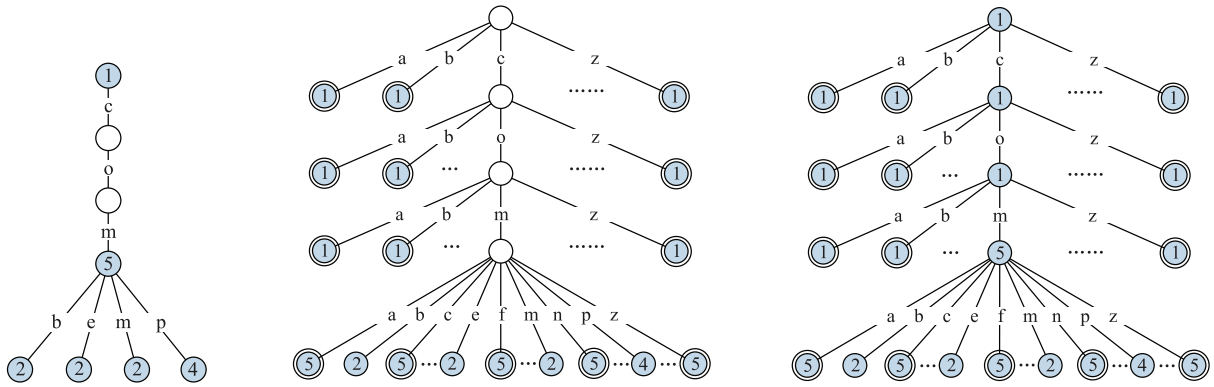
Pass Three may choose a final next hop from a set of candidate next hops, so CONSERT may produce many different output routing tables for a given input one. Fig. 5 just shows one of them. But CONSERT ensures that all the output tables are optimal (they all have the same number of prefixes). The appendix contains a mathematical formulation of CONSERT and a formal proof for its optimality.

Algorithm 3 Pass Three.

```
1:  procedure PassThree(Trie trie)
2:      for each node v in trie (from root to leaves) do
3:          if v ≠ trie.root and v.inherit_next_hop() ∈
    v.next_hop then
4:              v.next_hop ← NULL;
5:              w ← v.parent_node;
6:              w.S_w ← w.S_w ∪ v.component;
7:              if v is leaf node then
8:                  delete v;
9:          else
10:             p = RandSelect(v.next_hop);
11:             ▷ randomly pick up p from v.next_hop
12:             v.next_hop ← p;
```

Algorithm 4 The adapted LPM algorithm for the optimal trie.

```
1:  procedure LPM_Lookup(Node* root, string name)
2:      if root = NULL then
3:          return NULL;
4:      cur_node ← root;
5:      next_hop ← root.next_hop;
6:      for each component c_i in name do
7:          if cur_node.child[c_i] ≠ NULL then
8:              ▷ match
9:              cur_node ← cur_node.child[c_i];
10:             if cur_node.next_hop ≠ NULL then
11:                 next_hop ← cur_node.next_hop;
12:         else if c_i ∈ S_cur_node then
13:             return next_hop;
14:         else
15:             if cur_node.child['#'] ≠ NULL then
16:                 cur_node ← cur_node.child['#'];
17:                 next_hop ← cur_node.next_hop;
18:             return next_hop;
19:     if cur_node.child['#'] ≠ NULL then
20:         next_hop ← cur_node.child['#'].next_hop;
21:         return next_hop;
```



(a) Character trie for a sample routing table.

(b) Character trie after Pass One.

(c) Character trie after Pass Two.

Fig. 6. Character trie representation.

2.5. The adapted LPM algorithm

The LPM algorithm on the optimal trie representation differs from convention in two ways: (a) Because of the '#' nodes introduced, a prefix's next hop may be inconsistent before and after compression. E.g., prefix `/google` has a next hop of 2 in Fig. 5, while in the original routing table its next hop is 4. Another example is `/yahoo/sports`. The final next hop of these prefixes are actually contained in their '#' child nodes. This case corresponds to the situation that the searched name equals the prefix represented by an internal node in the trie. Therefore, after the match of a whole prefix, we look ahead to see if the last matched node has a '#' child node. If so, return the next hop of the '#' node (line 19 to 21 in Algorithm 4). (b) Due to the set S_v on node v. When a mismatch of component c on node v occurs, the follow-up operation depends on if c belongs to S_v. If $\mathtt{c} \in \mathtt{S_v}$, then return the next hop of v (if empty then return v's inherited next hop) (line 12–13 in Algorithm 4); otherwise (line 14 in Algorithm 4), before the lookup process terminates we also look ahead to see if the last matched node has a '#' child node and return its next hop (line 15–18 in Algorithm 4).

The rest node transition and termination principle adheres to the conventional LPM. Such adapted LPM algorithm on the optimal trie is illustrated in Algorithm 4, where `child` is a map from a component to a node's corresponding child node.

2.6. Character trie

Faced with the requirement of storing set S_v at node v, we propose to represent the routing table by a character trie. The character trie representation of a simple routing table is illustrated in Fig. 6(a). Each node in the character trie has at most 26 (number of characters in the alphabet) children nodes, therefore, we adjust the CONSERT algorithm accordingly.

Pass One: for any non-leaf node v in the trie, create a child node for each alphabet character that is originally not v's child, and push the next hop information of node v (or v's closest non-empty ancestor) down to the newly created children nodes, as illustrated in Fig. 6(b). By this means, Pass One supplements all the "missing" children nodes so that each non-leaf node has 26 children. Pass Two remains the same, and the character trie after Pass Two is shown in Fig. 6(c). When Pass Three removes some leaf nodes, a node v does not need the set S_v to record the characters corresponding to those removed children nodes.

We can see that the trie after Pass One becomes very "fat" since many children nodes are created. Obviously, the next hop of each parent node is copied for a lot of times, therefore, the `v.PrevalentSelect()` function in Pass Two will always intend to select that next hop as the candidate next hop for the parent node. The result is that all the newly created children nodes are eventually removed in Pass Three, which means the character trie has very little opportunity to reduce the number of prefixes. Hence, the output routing table would be very similar to the input one. In our example, the output optimal trie exactly equals the original one, as illustrated in Fig. 6(a). If we adopt component trie to construct this routing table, the optimal output is shown in Fig. 7. The number of prefixes is reduced from 6 to 4, revealing that the component trie is superior to character trie for routing table compression.

2.7. Binary trie

The third way to represent a routing table is using a binary trie. We translate each name prefix in the routing table into its ASCII code (excluding the delimiters), and then build a binary trie based on these bit strings. Still taking the routing table in Fig. 6(a) as an example, its binary trie representation is illustrated in Fig. 8. Different from the binary trie for IP routing table, this trie has unbounded depth. This time, CONSERT degrades to the ORTC [30] algorithm. After three passes, the output optimal routing table is the same as the input one in Fig. 8.

An important property of the binary trie for the name-based routing table, compared with that for the IP routing table, is that it has *sparse* solid nodes. Therefore, ORTC may have limited opportunity to optimize the number of prefixes for name-based routing tables.

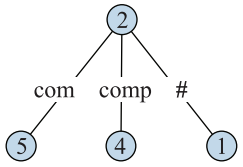


Fig. 7. The optimal trie based on the component trie.

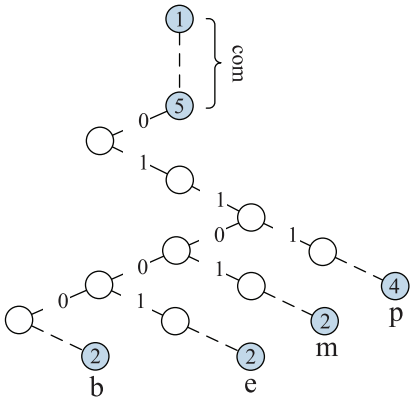


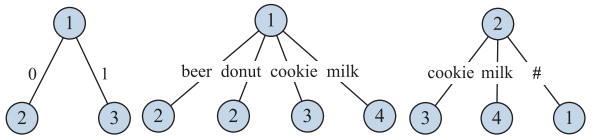
Fig. 8. Binary trie representation.

3. Analysis and improvements

3.1. Analysis – limited address space vs. unlimited name space

IP addresses are of fixed length – 32 bits or 128 bits, thus an IP routing table has a limited address space – 2^{32} or 2^{128}. This limitation is reflected by the binary trie in two aspects: (1) each node has a outbound degree of at most 2, (2) the depth of the trie is at most 32 or 128. Names, on the other hand, possess an unlimited namespace because they have unlimited lengths, and each level has unlimited number of components. They are reflected in the component trie of name-based routing table by two features: (1) each node can have countless children, (2) the depth of the trie is unlimited. We believe these are the fundamental reasons that make the optimal routing table construction algorithms different for IP and name-based routing tables. Below, we study how these distinctions impact the algorithm.

Fig. 9 (a) and (b) shows two brief tries for a simple IP routing table and a simple name-based routing table, respectively. Both routing tables have a default route. In essence, the default in Fig. 9(a) is generally unnecessary because the two children nodes have covered all the scenarios in the address space: if the current input bit is 0, go to the left branch; otherwise go to the right branch. The parent node does not need a default next hop for the uncovered scenarios. However, the case in Fig. 9(b) can really happen, and is actually the common case. Because no matter how many children the parent node has, they still cannot cover all the scenarios in the name space, and those uncovered scenarios require the parent node to install a default next hop for them. E.g., the `coffee` component does not match any branch, so it will take the parent node's next hop. And for exactly this reason, CONSERT needs to create a '#' child to deal with the uncovered situations. When we apply CONSERT to the trie in Fig. 9(b), the output optimal trie is illustrated in Fig. 9(c), where components `beer` and `donut` are removed. Actually this is the cost of our algorithm – some information of the original routing table is missing. (The reason why ORTC for the IP trie does not lead to any information missing is that the node in IP trie has at most 2 children, deleting a child node means its next hop can be delegated by its parent node's.) Therefore, set S_v is required as an auxiliary data structure to recover this information during the lookup. For the root node, $S_v = \{\texttt{beer, donut}\}$.



(a) Simple IP trie. (b) Simple name-based trie. (c) Optimized name-based trie.

Fig. 9. Compare IP trie and name-based tries.

Now let's see how this information missing leads to the confuse when looking up a name on the optimal trie. Take the same example of `coffee`, obviously it cannot match neither `cookie` nor `milk`, but we are not sure if it can match '#' only depending on the information on the trie. If `coffee` matches '#', then its next hop is 1, otherwise 2. To remove this uncertainty, we need the help of S_v. We know that `coffee` $\notin S_v$, so `coffee` $\in$ '#', thus `coffee` can match '#' and its next hop is 1. And because `beer` $\in S_v$, so `beer` $\notin$ '#', thus it does not match any child and its next hop is 2. This analysis reveals the main reason to adapt the LPM in Algorithm 4.

3.2. Improvements

Next we provide three improvements to CONSERT.

1. Removing the default routeassumption

As formulated above, CONSERT assumes that the input routing table always has a default route, i.e., a next hop for the "$*$" prefix. The presence of the default route ensures that every newly created child node inherits a next hop in Pass One. However, in the real world we often encounters *default-free* routing tables, such as the routing tables in the backbone routers. A simple and effective approach to deal with default-free routing tables is to introduce dummy default route: assign an invalid next hop, say -1, to the "$*$" prefix. After Pass Three, the invalid next hops in the trie are removed, and corresponding nodes are labeled as empty. In this way, we can remove the default route restriction.

2. Removing the single next hop assumption

CONSERT also assumes that each prefix in the initial routing table has a single next hop, while in real routing tables multiple next hops are often found. Especially, multiple next hops are common in NDN because name-based routing protocols like NLSR [24] provides multiple hops for a prefix by default. There are multiple methods to overcome this limitation in CONSERT.

First, we can pick a single best next hop, if possible, from the set of next hops for a prefix by some metric, then we can apply the CONSERT algorithm. If more than one next hops

tie for best, then we allow the input routing table to CONSERT to have multiple next hops for a prefix, so we slightly modify Pass One to – a new child node may inherit multiple next hops from its ancestor. Pass Two and Pass Three remain the same. Actually, multiple next hops can give CONSERT more opportunities to achieve better compression. E.g., if a node v has 4 children nodes, when each prefix can only have one next hop, the first 2 children nodes have a best next hop of i, and the rest 2 children nodes have a best next hop of j. Either i or j is selected for node v during Pass Three, only two children nodes can be set to empty. However, when each prefix is allowed to have multiple next hops, and v's all children nodes coincidently have best next hops of i and j, either i or j is selected for node v during Pass Three, all the 4 children nodes can be set to empty.

Second, if we cannot select best next hop(s) for a prefix, we allow the input routing table to CONSERT to have all the original next hops for a prefix, the modification to CONSERT is the same as the the method above. This method does not distinguish all the next hops for a prefix, but treat them equally. These two methods do not preserve the multiple next hop information in the input routing table.

Third, if the multiple next hop information needs to be preserved "as-is", we can still apply CONSERT. We take the set of multiple next hops for a prefix as an *atomic* (*indivisible*) entity, or a *virtual* next hop that represents the set of next hops of a prefix. By this means, CONSERT manipulates the sets of sets of next hops, rather than sets of next hops.

3. Minimize the number of '#'

The presence of '#' in the optimal trie requires an associated set S_v at its parent node v to store the removed children components of node v. But we have opportunities to reduce the number of '#' so as to decrease the memory cost of S_v. Assume the next hop of a '#' node is p. During Pass Three, while selecting an element from P_v as the next hop for v, rather than random selection, we give priority to p. If $p \in P_v$, then p is selected as v's next hop so that the '#' child node will be removed, as well as the associated set S_v. After applying this method to all the '#' nodes, the number of '#' is minimized.

4. Handling updates

After optimal routing table is constructed, prefixes may need to be inserted or removed from the routing table due to routing changes or content publishing and withdrawal. This section deals with updates – prefix insertions and deletions – on the optimal routing table. Of course we can commit the updates to the original routing table and build an optimal one afterwards, which is, however, inefficient. Below we describe the algorithm to handle updates on the optimal trie.

4.1. Insertion

Inserting a prefix into an original trie is a trivial process: go downwards the trie from the root until the prefix component cannot match an edge, then create nodes and edges for the rest components in the prefix. But since we have introduced the '#' symbol, the insertion on an optimal routing trie is a little complicated than this process. We list the following three cases for an insertion on the optimal trie.

Case 1: The simplest case: no '#' symbol is met while inserting the prefix, just adopt the conventional insertion algorithm. E.g., inserting prefix `/google/ad` with next hop 6 to the optimal trie in Fig. 5. The component `ad` requires a new node as the child node of node 2, so we create a new node and set its next hop to 6. In this way, the component `ad` is added to the children component set of node 2.

Case 2: Insert a prefix whose parent prefix have p been *optimized*, i.e., the node corresponds to prefix p has been set to empty or removed. E.g., inserting prefix `/google/news/domestic` with next hop 7. `/google/news/` is the prefix of this new prefix, and it has been *optimized* since the node for `news` in the optimal trie has been removed. This case requires recreating a child node for `news`, leaving it empty, and continuing to create a descendant node for `domestic`, setting 7 as its next hop.

For the both cases above, after prefix insertion(s) the optimality of the trie may be affected, e.g., if node 2 has more than 3 children nodes whose next hops are 3 after several insertions, then the next hop of node 2 should be updated to 3, and nodes for components `scholar`, `news` and `image`, as well as their next hops, should be restored (corresponding to Pass Two and Three). This change may further affect the optimality of its parent node, so on and so forth. Therefore, we need to run Pass Two and Three on the trie after prefix insertion to keep its optimality. We recommend that they are re-run periodically rather than after each insertion to save CPU resource.

Case 3: Insert a prefix who has a path in the original trie, but the end node of the path is empty. E.g., inserting prefix `/yahoo/sports` with next hop 8. The last node of path `/yahoo/sports` is empty in the original routing table (Fig. 1). In the optimal trie, the actual node corresponding to prefix `/yahoo/sports` is the '#' child node (node 10) of node 8, rather than node 8 itself (refer to Algorithm 4). For this case, just replace node 10's next hop by 8. Note that if the new next hop equals that of a sibling node of node 10, say the `nfl` node, we set the `nfl` node to empty. If the `nfl` node is a cleaf node, remove this node. The reason is under this scenario the prefix `/yahoo/sprots/nfl` is no longer needed, because its next hop can be delegated by its parent prefix `/yahoo/sprots`, corresponding to node 10 in Fig. 5. This case will not affect the optimality of the optimal trie. The insertion algorithm is presented in Algorithm 5.

It's worth pointing out that, if the inserted prefix already exists in the routing table, the insert operation means updating the next hop of the prefix.

4.2. Deletion

Prior to delete a prefix, we always assume that the prefix to be removed does exist in the routing table. According to the lookup result, there are three case to consider: (1) The lookup process involves neither set S_v nor the '#' symbol, just set the last matched node to empty. If it is a leaf node, delete the node. (2) The last component matches set S_v, delete the component from S_v. (3) The last matching node is the '#' node, delete the '#' node. The delete operation is presented in Algorithm 6.

Algorithm 5 The insertion algorithm for the optimal trie.

```
1:  procedure Insert(Node* root, string prefix, next_hop p)
2:      if root = NULL then
3:          return NULL;
4:      cur_node ← root;
5:      next_hop ← root.next_hop;
6:      for each component c_i in prefix do
7:          if cur_node.child[c_i] ≠ NULL then        ▷ match
8:              cur_node ← cur_node.child[c_i];
9:              if c_i is the last component then
10:                 if cur_node.child['#'] ≠ NULL then   ▷ case 3
11:                     cur_node ← cur_node.child['#'];
12:                     cur_node.next_hop ← p;
13:                     set any sibling node of cur_node whose
14:                     next hop equals p to empty.
15:                 else
16:                     cur_node.next_hop ← p
17:                 return;
18:         else                                  ▷ case 1 and case 2
19:             if c_i ∈ S_cur_node then
20:                 ▷ operation required by case 2
21:                 S_cur_node ← S_cur_node − c_i
22:             create a new child note w for component c_i;
23:             cur_node ← w;
24:             if c_i is the last component then
25:                 cur_node.next_hop ← p
26:                 return;
```

Algorithm 6 The deletion algorithm for the optimal trie.

```
1:  procedure Delete(Node* root, string prefix)
2:      if root = NULL then
3:          return NULL;
4:      cur_node ← root;
5:      next_hop ← root.next_hop;
6:      for each component c_i in prefix do
7:          if cur_node.child[c_i] ≠ NULL then        ▷ match
8:              cur_node ← cur_node.child[c_i];
9:              if c_i is the last component then
10:                 if cur_node.child['#'] ≠ NULL then   ▷ case 3
11:                     cur_node ← cur_node.child['#'];
12:                     remove node cur_node;
13:                 else                              ▷ case 1
14:                     cur_node.next_hop ← NULL;
15:                     if cur_node is leaf node then
16:                         remove node cur_node;
17:                 return;
18:         else                                      ▷ case 2
19:             if c_i ∈ S_cur_node then
20:                 S_cur_node ← S_cur_node − c_i
21:             return;
```

Table 2
Hardware configuration.

Item	Specification
CPU	Intel Xeon E5645 × 2 (6 cores × 2 threads, 2.4 GHz)
RAM	DDR3 ECC 48GB (1,333MHz)
Motherboard	ASUS Z8PE-D12X (INTEL S5520)

5. Evaluation

This section thoroughly evaluates the CONSERT algorithm in terms of the compression ratio, time cost, etc. The performance of the joint method of CONSERT+NameFilter is also examined in terms of the lookup speed and the memory cost.

5.1. Experiment platform

The experiments are conducted on a commodity server platform, running OS Linux 2.6.43. Platform hardware configuration is listed in Table 2. The CONSERT algorithm, as well as the NameFilter method, is implemented by the C++ programming language, using OpenMP [32] to support multi-thread programming.

5.2. Generating name-based routing tables

Since there is no public name-based routing tables available, we generate synthetic ones for our experiments. The routing table is generated in this way: domain names are at first collected by a web crawler, then they are hierarchically reversed into NDN-style prefixes, e.g., www.journal.com is transformed to `/com/journal/www`. Next we map each domain name to an IP address by querying DNS. Afterwards, we obtain the next hop number by looking up the IP address against an IP routing table downloaded from `archive.routeviews.org`. If a prefix maps to multiple next hops, then the first one is picked. Using these prefixes and corresponding next hops we build a basic NDN routing table. Subsequently, we generate synthetic prefixes in this way: for each prefix in the basic routing table, we randomly generate a number of components and append each component to the prefix, so that this prefix is expanded to multiple longer prefixes. Their next hops are assigned in two ways: inheriting from ancestor nodes (excluding parent nodes) or by manual generation (so that the next hop popularity among a node's children can be tuned). Then the new prefixes are inserted into the basic routing table. Afterwards, we repeat this "expansion" process on the new prefixes, until a preset number of totaly prefixes is reached. At last we obtain 4 name-based routing tables consisting of 1000, 10,000, 100,000 and 1,000,000 prefixes, respectively. The distribution of prefix length and the number of components each prefix contains is presented in Fig. 10.

The next hops are tuned to obey two cases: (1) common case. There are some popular next hops among a router's all the ports, so a prefix is more probable to be assigned these next hops. This is common for a router because the connectivity for some ports are higher than the rest ones, which means these ports can reach more networks. (2) limit case. Each next hop has almost equal popularity, e.g., the next hops

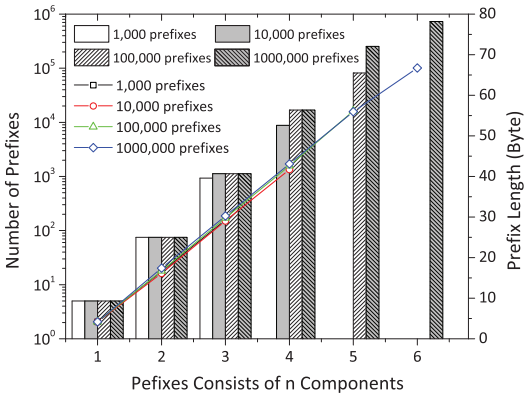


Fig. 10. Prefix length and component number distribution.

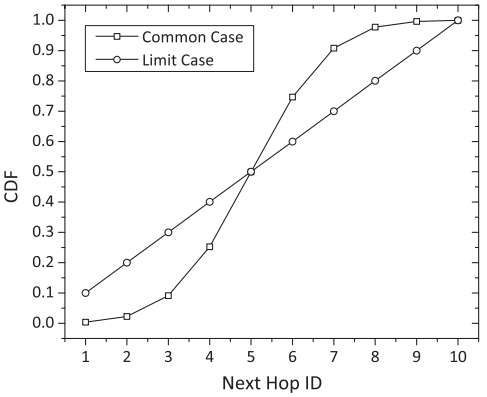


Fig. 11. Next hop popularity distribution.

are uniformly distributed. More generally, if we define the *popularity* of a next hop i (denoted by p_i) as the number of its occurrences divided by the total occurrences of all the next hops, the we can say $\boldsymbol{p} = \{p_1, p_2, ...\}$ is the *popularity distribution* of the next hops. The next hop popularity distribution for these two cases are shown in Fig. 11, where the next hops 4, 5, 6 and 7 are popular ones for the common case.

5.3. Compression results of single next hop

This section presents the compression results of CONSERT. Tables 3 and 4 list the experimental results for the common case and the limit case respectively. In both tables, the 2nd and 3rd columns show the number of prefixes in the synthetic routing table before and after compressing, the 4th column reports the compression ratio in terms of the reduced prefix numbers. These three columns reveal that CONSERT achieves the best compressibility with the component trie, which could reach around 45% for the common case and 18% for the limit case. Character trie and binary trie achieves much smaller compressing ratio, and they have the same compressibility in our experiments. Comparison between the common case and the limit case reveals that, the benefit of CONSERT stems from the high popularity of next hops. If we rank the popularity in $\boldsymbol{p}$ from high to low, a skewed popularity distribution (e.g., the common case in this paper) can lead to higher compression ratio by CONSERT than the balanced popularity distribution (e.g., the limit case in this paper).

Other than the number of prefixes, the number of nodes in the tries are also significantly reduced (column 5 and 6 in Tables 3 and 4). CONSERT achieves the most node reduction as well, which is the direct evidence that CONSERT diminishes the memory consumption of tries.

The number of '#' symbol using random selection (`RandSelect()` in Pass Three) in the optimal component trie is listed in column 7. Since each '#' requires memorizing a set of components, we want to keep the number of '#' as small as possible. Column 8 of both tables report the minimal number of '#' using the improvement method in Section 3.2. By using some compact data structures, such as Bloom filter, we can keep the memory consumption of component set S_v required by the '#' nodes very small.

The last column reports the processing time on our experimental platform, which reveals that component trie is the most time-efficient (only around 2000 μs), while character trie consumes the most time cost, reaching as long as 470 seconds!

5.4. Compression results of multiple next hops

Next we examine CONSERT's performance when a prefix may have multiple next hops in the input routing table. The method to build a synthetic routing table with multiple next hop information is similar to building one with single next hop, the distinction is we preserve the multiple next hops if a prefix maps to multiple ones in the downloaded IP routing table, rather than pick up the first one. Hence, while expanding the short prefixes to longer ones, a new prefix may inherit a set of next hops, or is assigned with a set of next hops selected from all the unique next hop sets in the routing table. We let the distribution of the next hop sets obey the common case.

CONSERT preserves the multiple next hop information by treating each individual set of next hops as a unique atomic next hop set. Table 6 lists the results obtained using this method. Surprisingly, we find that CONSERT still achieves good compression. It reduces roughly 30% of the prefixes in the routing tables. The reason is though there could be enormous potential unique next hop sets, in real routing tables such next hop sets in use are limited. This result can be affected by the multi-path routing strategy in use, which assigns next hop(s) to each prefix.

5.5. Memory consumption reduction

As aforementioned, CONSERT can be used together with other routing table compressing algorithms. As an example, we first apply CONSERT to obtain the optimal routing table. Afterwards, NameFilter [16], a two-stage-Bloom-filter method, is applied to the optimal routing table to further expedite the lookup process and reduce memory consumption. The NameFilter method organizes routing table by Bloom filters to save memory cost and achieve high-speed LPM lookup. Prefixes are divided according to their levels, with all prefixes of a given level stored in the same Bloom filter.

Table 3
Experimental results on synthetic routing table (single next hop, common case).

	No. of prefixes before compress	No. of prefixes after compress	Compression ratio (%)	No. of trie nodes before compress	No. of trie nodes after compress	No. of '#'	Min No. of '#'	time (ms)
component	1000	584	41.60	1011	607	38	38	2
trie	10,000	5522	44.78	10,005	5751	421	405	24
	100,000	54,930	45.07	99,980	57,166	4354	4200	273
	1,000,000	548,584	45.14	999,782	570,669	43,100	41,601	2841
char	1000	824	17.60	288,142	9477	–	–	292
trie	10,000	8118	18.82	3,139,836	95,581	–	–	3055
	100,000	81,887	18.11	33,895,940	963,101	–	–	34,935
	1,000,000	815,495	18.45	478,516,302	9,595,675	–	–	470,500
binary	1000	824	17.60	86,474	72,034	–	–	133
trie	10,000	8118	18.82	875,569	727,365	–	–	1607
	100,000	81,887	18.11	8,777,500	7,328,347	–	–	20,969
	1,000,000	815,495	18.45	87,661,413	73,012,929	–	–	231,171

Table 4
Results on synthetic routing table (single next hop, limit case).

	No. of prefixes before compress	No. of prefixes after compress	Compression ratio (%)	No. of trie nodes before compress	No. of trie nodes after compress	No. of '#'	Min No. of '#'	time (ms)
component	1000	823	17.70	1011	836	56	53	2
trie	10,000	8187	18.13	10,010	8319	526	475	24
	100,000	81,842	18.16	99,980	83,030	5064	4516	287
	1,000,000	817,670	18.23	999,782	830,085	50,706	45,696	2943
char	1000	917	8.30	12,013	10,871	–	–	294
trie	10,000	9055	9.45	115,488	105,660	–	–	3153
	100,000	89,913	10.10	1,154,143	1,047,584	–	–	34,029
	1,000,000	899,824	10.02	11,538,820	10,486,241	–	–	465,405
binary	1000	917	8.30	91,398	82,719	–	–	140
trie	10,000	9055	9.45	877,530	803,430	–	–	1476
	100,000	89,913	10.10	8,770,218	7,965,429	–	–	21,328
	1,000,000	899,824	10.02	87,680,190	79,734,096	–	–	235,168

Table 5
Memory consumption of routing tables before and after compression.

No. of prefixes	Memory consumption before compress (MB)	Memory consumption & compression ratio after compression					
		CONSERT (MB)	Compression ratio (%)	NameFilter (KB)	Compression ratio (%)	CONSERT+NameFilter (KB)	Compression ratio (%)
1000	0.116	0.0782	32.31	29.952	74.09	14.309	87.62
10,000	1.145	0.7602	33.60	284.976	75.11	133.141	88.37
100,000	11.442	7.5728	33.81	2,736.116	76.09	1,276.018	88.85
1,000,000	114.416	75.651	33.88	27,357.324	76.09	12,750.778	88.86

Table 6
Compression results when preserving multiple next hop information.

No. of prefixes before compress	No. of prefixes after compress	compression ratio (%)
1000	702	29.80
10,000	6954	30.46
100,000	68,951	31.05
1,000,000	680,781	31.92

Meanwhile, prefixes are also divided by their next hop port numbers, and all the prefixes with the same port number are stored in the same Bloom filter. Therefore, there are two sets of Bloom filters, and each prefix is inserted twice. We then perform the Longest Prefix Match to obtain the matched prefix for a given name, by querying Bloom filters according to prefix levels: from the Bloom filter with the longest level to the one with the shortest level. A match on a Bloom filter implies that the longest matched prefix is of current level. Afterwards, the matched prefix is queried in all the port Bloom filters, and a match on a Bloom filter reveals the next hop number. This section examines the memory consumption of

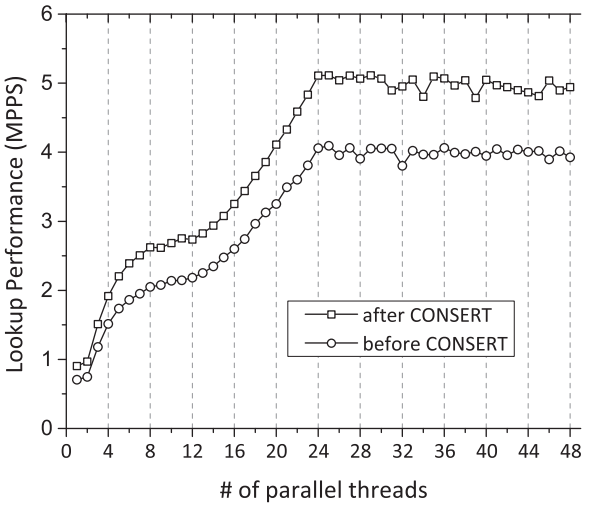


Fig. 12. Multi-thread lookup throughput based on component trie.

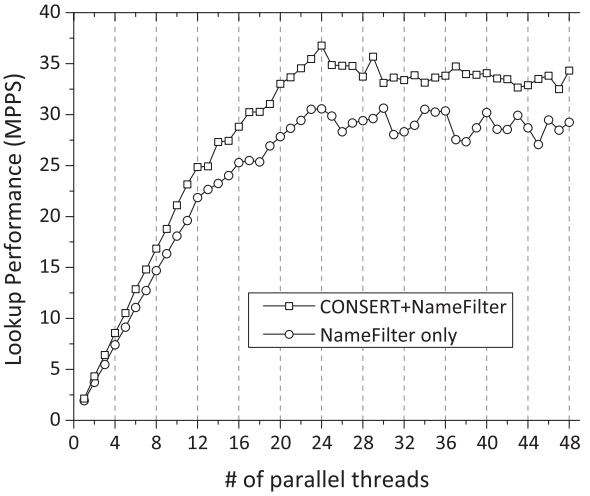


Fig. 13. Multi-thread lookup throughput based on NameFiler alone and CONSERT+NameFilter.

both individual methods as well as the joint method. The results are presented in Table 5, note that we only list the results for the common case next hop distribution. Table 5 reveals that CONSERT and NameFilter individually obtains a compression ratio of around 34% and 76%, respectively. However, CONSERT+NameFilter achieves a compression ratio of roughly 88%. Since Bloom filters can not handle updates because they can not perform deletions, it is appealing that CONSERT deals with updates to the routing table, and NameFilter generates a compact forwarding table accordingly.

5.6. Lookup performance

This section provides the lookup performance of CONSERT (based on the component trie structure), as well as the joint method of CONSERT+NameFilter. Firstly, we measure the lookup performance before and after CONSERT is applied using multiple treads. Because CONSERT operates on the component trie, the lookup results are also derived based on the component trie, which are illustrated in Fig. 12. For single thread, the lookup speed increases from 0.71 million packet per second (MPPS) to 0.90 MPPS, leading to 21.11% increase. The highest throughput by multi-thread increases from 4.09 MPPS to 5.11 MPPS, leading to 24.93% increase. The benefit is due to less and shorter prefixes, so the outbound degrees of the nodes and the depth of the tree are reduced. Hence, each lookup has a shorter path to go through.

Fig. 12 shows that lookup performance increases almost linearly and then flattens out at 24 threads, the reason is that the CPU in our experiment platform has 24 hardware threads (refer to Table 2). If more than 24 threads are used, the threads will compete for the hardware resource, and the number of concurrent threads is 24 at any time in the experiment. Therefore, more threads than 24 will not help to improve the lookup performance anymore. The behavior of the curves in Fig. 13 can also explained by this reason.

Next we examine the lookup performance of the joint method of CONSERT+NameFilter. The curve labeled by square symbols in Fig. 13 shows the lookup performance of CONSERT+NameFilter. Recall that the first stage of the NameFilter method assigns a Bloom filter to prefixes with the same length (in terms of the number of components). Since the prefixes are shortened by CONSERT, the number of Bloom filters required is also reduced. This decreases the number of memory references while looking up a name, and hence increases the lookup throughput. As a comparison, Fig. 13 also present the lookup performance when only NameFilter is performed, denoted by the curve with circle symbols. With the joint method, the highest throughput of multi-thread reaches 36.78 MPPS, while NameFilter alone achieves 30.63 MPPS. Hence, the joint method obtains 16.72% improvement.

5.7. Update performance

The update performance is measured by continuously inserting and deleting prefixes on the optimal routing table. Experiments show that the insertion speed can achieve 408K prefixes per second (K/s), and the deletion speed is 375 K/s.

6. Related work

In the literature, routing table compression generally refers to compressing IP routing tables, and there has been a wide range of research works devoted into this problem. We cannot fully cover such a vast background but only list the most notable ones. ORTC [30] was proven to be the theoretically optimal compression algorithm in terms of the number of prefixes, and it remains the best algorithm in terms of its compression ratio since 1999. Experiments on real backbone routing tables also showed superb performance of ORTC. It is worth pointing out that ORTC aims at reducing the number of prefixes at maximum, rather than elaborating on compressing the data structures, so ORTC can be applied jointly with other data structure compressing techniques to further reduce the memory requirement, e.g., the fast IP forwarding structure presented in [33]. In [33], the main technique is the controlled prefix expansion, which transforms a set of prefixes into an equivalent set with fewer prefix lengths. In addition, the authors use optimization techniques based on

dynamic programming, as well as local transformations of data structures to improve cache behavior. When applied to trie search, their techniques provide a range of algorithms (Expanded Tries) whose performance can be tuned. Other notable prior efforts that provide data structure compressing methods include Lulea [34], LC-trie [35], Bitmap trie [36] and FlashTrie [37]. Lulea [34] uses very little memory, averaging 4–5 bytes per entry for large routing tables. This small memory footprint allows the entire data structure to fit into the processor's cache, thus speeding up the operations. LC-trie [35] compresses the IP-routing table from another angle – it is a trie structure that combines path compression (Patricia tree [38]) and level compression [39]. This data structure enables us to build efficient, compact, and easily searchable implementations of an IP-routing table. Bitmap trie [36] compresses non-leaf-pushed multibit tries, where each node contains two bit maps, one for internally stored prefixes and one for the external pointers. FlashTrie [37] combines hash operation and multibit-trie compressing structures to reduce off-chip memory accesses, thus achieving high lookup throughput.

Name-based routing is not a new concept, but has been studied in the literature [19–26]. Name-based routing also needs a routing table to route packets to their destinations, where each table entry is filled with name prefixes, rather than IP prefixes. For name-based routing table, several compressing techniques with compact data structures have been proposed, often along with fast lookup purposes. NCE [27,28] employs a component-trie to organize the name-based routing table. In the trie the name components are represented and replaced by a unique code (integer), reducing memory cost and accelerating lookup. NameFilter [16] divides prefixes in the name-based routing table into groups by their lengths as well as next hops, and employs Bloom filters to store each group separately. BFAST [40] employs a Counting Bloom filter to balance the load among hash table slots, making the number of prefixes in each nonempty slot close to 1, and thus enabling high lookup throughput and low latency. Moreover, the First-Rank-Indexed technique is proposed to effectively reduce the massive storage requirement for the pointers in all the hash table slots. The memory-efficiency of Bloom filter helps compress the name-based routing table. All these methods are orthogonal to the CONSERT method and can be applied jointly.

7. Conclusion

This paper proposes an algorithm called CONSERT to build an optimal name-based routing table with the minimal number of prefixes. CONSERT consists of three passes, where Pass One introduces the '#' symbol and pushes the next hops down to the '#' nodes, Pass Two pushes the most prevalent next hop(s) upwards as high as possible, and Pass Three determines the next hop for each prefix. CONSERT can apply to the situations of no default route and multiple next hops, and it works for tries of different granularities as well. We formulate the algorithm and prove its optimality using the induction method. Experimental results show that CONSERT can effectively reduce the number of prefixes in a routing table and improve the lookup performance, especially cooperates with other orthogonal data structure compressing methods like NameFilter.

Acknowledgment

This work is supported by 863 project (2013AA013502), NSFC (61202489, 61373143, 61432009, 61402254), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20131019172), and Jiangsu Future Networks Innovation Institute: Prospective Research Project on Future Networks (No. BY2013095-1-03). This work is sponsored by Huawei Technologies Co., Ltd.

Appendix

This section contains a mathematical formulation of CONSERT and a formal proof for its optimality. The proof proceeds via induction on the deepest level of the routing table trie.

A.1. Definitions

The proof is based on the component trie. For generality, CONSERT considers N-component prefixes for any integer N, where N is the deepest level of the component trie.

A formal definition of a routing table can be given by a function or a map. Let P_N be the set of all the prefixes with length (in terms of the number of components) less than or equal to N. Let A_N denote the prefixes of exactly N components, so $P_N = A_0 \cup A_1 \cup \cdots A_N$. Let H be the set of all possible next hops for the router, and Ω be the set of all the subsets of H. A routing table assigns a subset (may be empty) of H to all the prefix $x \in P_N$. So we define a *routing map* to be any map

$$R : P_N \to \Omega \tag{1}$$

We use $|x| = k$ to indicate x's length or level in the routing table's tree representation. We use $R(x)$ to denote the next hop of x in the routing map R, and say that x is *occupied* if $R(x) \neq \emptyset$. Let $|R|$ denote the number of occupied vertices in the tree, or the number of entries in the routing table. The root node in the tree on P_N is denoted as r_N. There is a unique path from the root r_N to every vertex x in P_N. On this path, either (1) there is a unique occupied vertex which is closest to x, but not equal to x, call it the ancestor of x, denoted by $Anc(x)$, or (2) there are no occupied vertices. We define the *inherited next hop* of x as

$$Inh[x, R] = \begin{cases} R(Anc(x)), & \text{in case (1)} \\ \emptyset, & \text{in case (2)} \end{cases} \tag{2}$$

Given two routing tables R_1 and R_2, for any prefix p in R_2, if $R_1(p) \subset R_2(p)$, then we say that $R_1 \subset R_2$ (read as R_1 covers R_2). If $\forall p \in P_N, R_1(p) = R_2(p)$, we say $R_1 = R_2$.

A.2. Formulate the algorithm

The algorithm presented below formulates CONSERT by symbols and formulas. The input is a routing map R after Pass One, and the output is a collection of optimally compressed routing tables. The algorithm proceeds in two steps. The first step pushes the prevalent next hops in the leaf nodes upwards the tree level by level, until all nodes in the tree are

occupied (correspond to Pass Two). The second step successively prunes the nodes until only the minimal number of entries remains (correspond to Pass Three). Given a vertex x in a component trie, we use $x_{c_1}, x_{c_2}, \ldots, x_{c_m}$ to denote its m children vertices (let $c_m =$ '#').

The input. The expected input for this algorithm is the output of Pass One. Now we assume thatthe input R is the result of Pass One on P_N, and therefore $R(r_N) = \emptyset$. **Step 1.** Define inductively a sequence of routing maps $(Q_0, Q_1, ..., Q_N)$ by

$$Q_N = R \tag{3}$$

and for $1 \le k \le N$:

$$Q_{k-1}(x) = \begin{cases} \texttt{PrevalentSelect}(Q_k(x_{c_1}), Q_k(x_{c_2}), ..., Q_k(x_{c_m})), \\ \qquad \text{if } x \in A_{k-1} \text{ and } Q_{k-1}(x) = \emptyset \\ Q_k(x), \text{ otherwise} \end{cases} \tag{4}$$

Q_0 will be used for step 2.
Step 2. Construct inductively a sequence of routing maps $(T_0, T_1, ..., T_N)$ by:

$$T_0(x) = \begin{cases} Q_0(x), & \text{if } x \neq r_N \\ \texttt{RandSelect}[Q_0(r_N)], & \text{if } x = r_N \end{cases} \tag{5}$$

And for $1 \le k \le N$:

$$T_k(x) = \begin{cases} T_{k-1}(x), & \text{if } x \notin A_k \\ \emptyset, \quad \text{if } x \in A_k \text{ and } Inh[x, T_{k-1}] \in T_{k-1}(x) \\ \texttt{RandSelect}[T_{k-1}(x)], \quad \text{otherwise} \end{cases} \tag{6}$$

Remarks for T_0: non-root vertices in T_0 inherit the same next hop informationfrom Q_0, while the root node randomly selects a next hop from $Q_0(r_N)$by $\texttt{RandSelect}[Q_0(r_N)]$.
Remarks for T_k, $(1 \le k \le N)$: For $x \notin A_k$, there are two cases to consider:

(1) $|x| < k$.In fact, the final next hop information for x has already been calculatedin T_{k-1}, now T_k inherits it from $T_{k-1}(x)$ directly;
(2) $|x| > k$. The final next hop information has been not calculatedyet, so $T_k(x)$ inherits *all* the possible next hops from $T_{k-1}(x)$(seemingly inherits from $T_{k-1}(x)$, but actually inherits from $Q_0(x)$).Both situations arrive at the same result: $T_k(x) = T_{k-1}(x)$, if $x \notin A_k$.

The output. The output of the above algorithm is the routing map T_N constructed atthe end of Step 2. Since many choices are made in Step 2 by the $\texttt{RandSelect()}$function, there are manypossible results. We denote them by $T_{R,s}$, where s is an index that distinguishesbetween them. The collection of all indices is a finite set S, so the possibleresults of the algorithm are the routing maps $\{T_{R,s}\}$ $(s \in S)$.

Remarks on the formulated algorithm:

(1) For $1 \le k \le N$, all the leaf vertices in $Q_k(x)$ are occupied, which means the routing information $Q_k(x)$ for each leaf node's corresponding prefix x is not empty. For non-leaf vertices, $Q_k(x)$ is the empty set if $|x| < k$, and is non-empty if $|x| \ge k$. Therefore, Q_0 has an entry for every prefix in P_N.
(2) Next we prove that $T_N \subset R$. For $1 \le k \le N$, $\forall x \in A_k$ $(i.e., |x| = k)$, we have $T_{k-1} = Q_0(x)$. This is because $|x| = k$, and in T_{k-1}, the next hop information of x has not been calculated yet, therefore $T_{k-1}(x) = T_0(x)$. According to Step 2, $T_0(x) = Q_0(x), x \neq r_N$, so $T_{k-1} = Q_0(x)$.

According to Step 2 again, $\forall x \in A_k$ $(|x| = k)$, $T_k(x)$ has two results:

(1) $T_k(x) = \texttt{RandSelect}[T_{k-1}(x)] \in Q_0(x)$
(2) $T_k(x) = \emptyset$, which means $Inh[x, T_k] = Inh[x, T_{k-1}] \in T_{k-1}(x) = Q_0(x)$.

Therefore, $T_k(x) \subset Q_0(x)$, $x \in A_k$, $1 \le k \le N$. Plus $T_0(r_N) = \texttt{RandSelect}[Q_0(r_N)] \in Q_0(r_N)$, yielding $T_0(x) = \texttt{RandSelect}[Q_0(x)] \in Q_0(x)$, $|x| = 0$, so we have $T_k(x) \subset Q_0(x), x \in A_k, 0 \le k \le N$.

On one hand, actually, $\forall x \in A_k$, $T_N(x_k) = T_k(x) \subset Q_0(x), 0 \le k \le N$ (also derived from Step 2), hence $T_N \subset Q_0$. On the other hand, Q_0 is derived from R, it has the same next hop information for each prefix x as R does (i.e., $\forall x \in P_k$, $Q_0(x) = R(x)$), therefore we say $Q_0 = R$. Hence $T_N \subset R$. Therefore, $\{T_{R,s}\} \subset R, s \in S$.

A.3. The theorem

Let R be any name-based routing table after Pass One, let R' be any routing map that covers R. Let $M_R = Q_0(r_N)$, where Q_0 is the result of Step 1. Note $|R|$ is the number of prefix entries in R. Then we have the following theorem.

The Optimality Theorem:

(1) $\boldsymbol{|R'| \ge |T_{R,s}|}$ **for all** $\boldsymbol{s \in S}$.
(2) **If** $\boldsymbol{R'(r_N) \notin M_R}$**, then** $\boldsymbol{|R'| \ge 1 + |T_{R,s}|}$ **for all** $\boldsymbol{s \in S}$.

This theorem implies that:

(i) $|T_{R,s}| = |T_{R,j}|$ for all $s, j \in I$. All the routing tables constructed by the algorithm have the same size.
(ii) $|T_{R,s}|$ is the smallest possible size for a routing table that covers R, meaning

that these routing tables achieve the optimal compression.

A.4. The proof

The proof of the *Optimality Theorem* relies on three operations that we call *merging*, *pushing* and *splitting* of routing tables. The merging operation takes m routing tables $R_{c_1}, R_{c_2}, ..., R_{c_m}$ on the P_N tree and joins them by creating a new root node to form a routing table $R_{c_1} * R_{c_2} * ... * R_{c_m}$ on the P_{N+1} tree. The new root node of the merged P_{N+1} tree is not occupied. The pushing operation takes a routing table R on the P_{N+1} tree and produces a new routing table $Push[R]$

on the P_{N+1} tree: the next hop of root r_{N+1} is assigned to the empty child node of r_{N+1}, then r_{N+1} is set to empty. The output routing table is $Push[R]$, such that $Push[R]$ can be written (uniquely) in the form $Push[R] = R_{c_1} * R_{c_2} * ... * R_{c_m}$ for some P_N trees $R_{c_1}, R_{c_2}, ..., R_{c_m}$. The notation $Push[R]$ indicates that we push down the routing information from the root to its children, which is a necessary step before we can split the table. Formally we have:

Merging: If x and y are $k-$component and $n-$component prefixes respectively, we denote by xy the $(k+n)-$component names obtained by concatenating x and y. Let $R_{c_1}, R_{c_2}, ..., R_{c_m}$ be routing maps on P_N. Define the routing map $R_{c_1} * R_{c_2} * ... * R_{c_m}$ on P_{N+1} by

$$R_{c_1} * R_{c_2} * ... * R_{c_m}(x) = \begin{cases} \emptyset, & \text{if } x = r_{N+1} \text{ (root node)} \\ R_{c_1}(y), & \text{if } x = c_1 y \\ \vdots & \\ R_{c_m}(y), & \text{if } x = c_m y \end{cases} \quad (7)$$

where $c_1, c_2, ..., c_m$ are name components corresponding to the edges that link $R_{c_1}, R_{c_2}, ..., R_{c_m}$ to the root node in $R_{c_1} * R_{c_2} * ... * R_{c_m}$ on P_{N+1}. $R_{c_1} * R_{c_2} * ... * R_{c_m}$ is the output of the merging operation.

For a routing map R after Pass One, its root has no next hop information ($R(r_N) = \emptyset$), therefore, it can be written uniquely in the form of $R = R_{c_1} * R_{c_2} * ... * R_{c_m}$ for some $R_{c_1}, R_{c_2}, ..., R_{c_m}$.

Pushing&Splitting: Let R be a routing map on P_N. Define a new routing map $Push[R]$ on P_N as follows:

$$\begin{aligned} &Push[R](r_N) = \emptyset \\ &Push[R](c_1) = \begin{cases} R(c_1), & \text{if } R(c_1) \neq \emptyset \\ R(r_N), & \text{if } R(c_1) = \emptyset \end{cases} \\ &\vdots \\ &Push[R](c_m) = \begin{cases} R(c_m), & \text{if } R(c_m) \neq \emptyset \\ R(r_N), & \text{if } R(c_m) = \emptyset \end{cases} \\ &Push[R](x) = R(x), \text{ if } x \in A_2 \cup A_3 \cup ... \cup A_N \text{ (i.e., } |x| \geq 2) \end{aligned} \quad (8)$$

The pushing operation pushes the next hop of the root to its empty children nodes. After R is pushed into $Push[R]$, $Push[R]$ can be written in the form $Push[R] = R_{c_1} * R_{c_2} * ... * R_{c_m}$, which means $Push[R]$ can be ***split*** into m routing maps $R_{c_1}, R_{c_2}, ..., R_{c_m}$ on P_{N-1}. This reveals that, while we have m routing maps on P_{N-1}, we can build a new routing map on P_N by the merging operation, and the new routing map is $Push[R]$, i.e., $Push[R] = R_{c_1} * R_{c_2} * ... * R_{c_m}$.

Some properties concerning $Push[R]$:

- **Property 1**: For any R, $|R| - Push[R] \in [-(m-1), 1]$.
- **Property 2**: For a routing table $T_{R,s}$ derived from the CONSERT algorithm, $|T_{R,s}| - |Push[T_{R,s}]| \in [-(m-1), 0]$. This is because the only situation when $|T_{R,s}| - Push[T_{R,s}] = 1$ is that the root node and all its children are occupied in $T_{R,s}$. However,the actual situation is only the root node will always be occupied in $T_{R,s}$, and therefore some (or all) children of it are empty.
- **Property 3** (*Important property): Let $R = R_{c_1} * R_{c_2} * ... * R_{c_m}$ be a routing map on P_{N+1} ($R_{c_1}, R_{c_2}, ..., R_{c_m}$ are routing maps on P_N), and let $S_R, S_{R_{c_1}}, S_{R_{c_2}}, ..., S_{R_{c_m}}$ be the index sets produced by the CONSERT algorithm for routing maps $R, R_{c_1}, R_{c_2}, ..., R_{c_m}$, respectively. For every $s \in S_R$, there exists unique $j_1 \in S_{R_{c_1}}, j_2 \in S_{R_{c_2}}, ..., j_m \in S_{R_{c_m}}$, such that $Push[T_{R,s}] = T_{R_{c_1},j_1} * T_{R_{c_2},j_2} * ... * T_{R_{c_m},j_m}$. $T_{R,s}$ is the optimally compressed routing map for R, and $T_{R_{c_1},j_1}, T_{R_{c_2},j_2}, T_{R_{c_m},j_m}$ are the optimally compressed routing maps for $R_{c_1}, R_{c_2}, ..., R_{c_m}$, respectively. This is the key property of the CONSERT algorithm – every compressed table constructed on the P_N tree is equivalent to a bunch of compressed routing tables on the P_{N-1} sub-trees, and these can be derived using the pushing and splitting operations.

Proof. The proof of the theorem is by induction on N – the number of the deepest levels in the tree. For $N = 1$, R is a routing map on P_1, the tree has $m+1$ vertices, namely the root r_1 and its m children. Let $p_1 = R(c_1), p_2 = R(c_2), ..., p_m = R(c_m)$ ($R(c_i)$ returns the next hop of one-component-prefix c_i in R), then $M_R =$ `PrevalentSelect`$(p_1, p_2, ..., p_m)$. Let $T_{R,s}$ be any routing table produced by the CONSERT algorithm with R as input, and R' be any routing table that covers R ($R' \subset R$). Let $G = R'(r_N)$ (note that G may be empty). Define

$$\Delta = |R'| - |T_{R,s}| \quad (9)$$

We must prove that $\Delta \geq 0$, and if $G \not\subset M_R$, then $\Delta \geq 1$. Because both $Push[R']$ and $Push[T_{R,s}]$ covers R, then $|Push[R']| = |Push[T_{R,s}]| = m$. Rewrite Δ as

$$\Delta = |R'| - |Push[R']| + |Push[T_{R,s}]| - |T_{R,s}| \quad (10)$$

We have proved that $|Push[T_{R,s}]| - |T_{R,s}| \geq 0$. For $|R'| - |Push[R']|$, there are three cases to consider: □

Case 1. $|R'| - |Push[R']| = 1$. Because here $|Push[T_{R,s}]| = m \geq |T_{R,s}|$, then $|Push[T_{R,s}]| - |T_{R,s}| \geq 0$, so $\Delta \geq 1$, it's done.

Case 2. $|R'| - |Push[R']| = 0$. Hence $\Delta = |Push[T_{R,s}]| - |T_{R,s}| \geq 0$. Since $|T_{R,s}| \in [1, m]$, we have two subcases to consider:

Subcase 1: $|T_{R,s}| = n \in [1, m-1]$, then $\Delta \geq 1$, so it's done.

Subcase 2: $|T_{R,s}| = m$, then $\Delta = 0$, we need to prove $G \subset M_R$ (remember that $G = R'(r_1)$). In this case, $p_1, p_2, ..., p_m$ all have a population of 1, which means $p_1 \cap p_2 \cap ... \cap p_m = \emptyset$, so in Step 2 `PrevalentSelect`$(p_1, p_2, ..., p_m) = p_1 \cup p_2 \cup ... \cup p_m$. Hence $M_R = p_1 \cup p_2 \cup ... \cup p_m$. Because here $|Push[R']| = m$, and $|R'| - |Push[R']| = 0$, so $|R'| = m$. Hence there are two scenarios for G to consider:

(a) $G = \emptyset$, then $G \subset M_R$, it's done.
(b) $G \neq \emptyset$, then $Push[R']$ pushes G to exactly one of root's children vertices in R'. So $G \subset R'(c_1)$ or $G \subset R'(c_2)$ or ... or $G \subset R'(c_m)$. Because $R' \subset R$, then $G \subset R(c_1) = p_1$ or $G \subset R(c_2) = p_2$ or ... or $G \subset R(c_m) = p_m$, which means $G \subset p_1 \cup p_2 \cup ... \cup p_m = M_R$, and it's done.

Case 3. $|R'| - |Push[R']| \in [-(m-1), -1]$. Since $|Push[R']| = m$, then $|R'| \in [1, m-1]$ ($|R'| = m$ has been considered in Case 2). Again, there are two subcases for $|R'|$ to consider.

Subcase 1: $|R'| = 1$. In this case $G \neq \emptyset$ and all the m children vertices of the root are empty. So $Push[R']$ assigns G to all the root's children vertices in R'. Again because $R' \subset R$, G is the most prevalent next hop among the children vertices and

has a population of m, meaning that $G \subset p_1 \cap p_2 \cap ... \cap p_m \neq \emptyset$. Meanwhile $M_R = \texttt{PrevalentSelect}(p_1, p_2, ..., p_m) = p_1 \cap p_2 \cap ... \cap p_m$, yielding $G \subset M_R$ and $|T_{R,s}| = 1$. Then $\Delta = 1 - m + m - 1 = 0$, it's done.

Subcase 2: $R' \in [2, m-1]$. In this case $G \neq \emptyset$ and at least 2 children vertices of the root in R' are empty. Denote by t ($t \in [2, m-1]$) the number of empty children vertices of the root in R'. Since the order of the children vertices makes no difference on the algorithm, we always assume the first t vertices are empty. Then $Push[R']$ assigns G to the first t children vertices of the root in R', again because $R' \subset R$, then $G \subset R(c_1) = p_1, G \subset R(c_2) = p_2, ..., G \subset R(c_t) = p_t$. Since $|R'| \geq 2$, $p_1 \cap p_2 \cap ... \cap p_m = \emptyset$ (if $p_1 \cap p_2 \cap ... \cap p_m \neq \emptyset$, then $|R'| = 1$). Therefore, $\texttt{PrevalentSelect}(p_1, p_2, ..., p_m)$ picks out the most prevalent next hop(s) among $p_1, p_2, \ldots, p_m$, and assign such next hop(s) to M_R. Assume that the next hop(s) in M_R has a population of q, i.e., for any next hop $u \in M_R$, q children vertices of the root contains u as next hop. Then three scenarios for G need to be considered:

(a) $G \not\subset M_R$: G is not among the most prevalent next hops, and we need to prove $\Delta \geq 1$. In this case $q > t$, so $q - t \geq 1$. Assume $u = \texttt{RandSelect}(M_R)$ and q children vertices will be filled with u in $Push[T_{R,s}]$, so $|T_{R,s}| = m + 1 - q$, yielding $|Push[T_{R,s}]| - |T_{R,s}| = m - (m + 1 - q) = q - 1$. While $|R'| - |Push[R']| = (m - t + 1) - m = -t + 1$, therefore $\Delta = (-t + 1) + (q - 1) = q - t \geq 1$.

(b) $G \subset M_R$ but $G \neq \texttt{RandSelect}(M_R)$: G is one of the most prevalent next hops but is not selected as the next hop for $T_{R,s}$, and we need to prove $\Delta \geq 0$. In this case $q = t$. Assume $u = \texttt{RandSelect}(M_R)$ and q children vertices will be filled with u in $Push[T_{R,s}]$, which yields $|Push[T_{R,s}]| - |T_{R,s}| = m - (m + 1 - q) = q - 1$. While $|R'| - |Push[R']| = -t + 1$, then $\Delta = (-t + 1) + (q - 1) = 0$.

(c) $G = \texttt{RandSelect}(M_R)$: $G \in M_R$ and G is selected as the next hop for $T_{R,s}$, so we need to prove $\Delta \geq 0$. In this case $q = t$, and t vertices will be filled with G in $Push[T_{R,s}]$, which yields $|Push[T_{R,s}]| - |T_{R,s}| = m - (m + 1 - t) = t - 1$, then $\Delta = (-t + 1) + (t - 1) = 0$.

The proof for $N = 1$ ends.

Next we prove the induction step, namely we assume our theorem holds for all integers less than or equal to N, and prove it for $N + 1$. So now R is a routing map on P_{N+1} output by Pass One, and $R = R_{c_1} * R_{c_2} * ... * R_{c_m}$, where $R_{c_1}, R_{c_2}, ..., R_{c_m}$ are routing maps on P_N. R' covers R ($R' \subset R$), and $Push[R'] = R'_{c_1} * R'_{c_2} * ... * R'_{c_m}$, where $R'_{c_1}, R'_{c_2}, ..., R'_{c_m}$ are routing maps on P_N. $T_{R,s}, T_{R_{c_1},j_1}, T_{R_{c_2},j_2}, ..., T_{R_{c_m},j_m}$ are optimally compressed routing maps for $R, R_{c_1}, R_{c_2}, ..., R_{c_m}$, respectively. Let $G = R'(r_{N+1})$, and $p_1 = T_{R_{c_1},j_1}(r_N), p_2 = T_{R_{c_2},j_2}(r_N), ..., p_m = T_{R_{c_m},j_m}(r_N), M_R = Q_0(r_{N+1})$.

A more intuitive view is illustrated in Fig. 14, which reveals that merging optimal routing maps on P_N can lead to an optimal routing table on P_{N+1}. Specifically, R_{c_i}, R'_{c_i} and $T_{R_{c_i},j_i}$ ($1 \leq i \leq m$), denoted by triangles, are routing maps on P_N. Here is the induction reasoning: if $T_{R_{c_i},j_i}$ is the optimally compressed routing map for R_{c_i}, then after the merging operation ($T_{R_{c_i},j_i}$'s are merged to $T_{R,s}$, R_{c_i}'s are merged to R), $T_{R,s}$ is also an optimally compressed routing map for R. (R and $T_{R,s}$ are on P_{N+1}.) In other words, if there exists any R' that covers R, then $|R'| \geq |T_{R,s}|$. Note that $R' \subset R$ yields $R'_{c_i} \subset R_{c_i}$, so it's easy to see $|R'_{c_i}| \geq |T_{R_{c_i},j_i}|$. We will use this condition in the induction method. Afterwards, we have the following induction procedure.

Induction hypothesis:

(1) $|R'_{c_i}| \geq |T_{R_{c_i},j_i}|$, $1 \leq i \leq m$;
(2) If $R'_{c_i}(r_N) \not\subset p_i$, $|R'_{c_i}| \geq 1 + |T_{R_{c_i},j_i}|$, $1 \leq i \leq m$

Goal:

(1) $|R'| \geq |T_{R,s}|$;
(2) If $G \not\subset M_R$, $|R'| \geq 1 + |T_{R,s}|$;

Define

$$\begin{aligned}\Delta &= |R'| - |T_{R,s}| \\ &= |R'| - |Push[R']| + |Push[R']| - |Push[T_{R,s}]| \\ &+ |Push[T_{R,s}]| - |T_{R,s}| \end{aligned} \tag{11}$$

We need to prove $\Delta \geq 0$, and $\Delta \geq 1$ if $G \not\subset M_R$. As aforementioned, there are unique routing maps $R'_{c_1}, R'_{c_2}, ..., R'_{c_m}$ and unique indices $j_1, j_2, \ldots, j_m$ such that:

$$\begin{aligned} Push[R'] &= R'_{c_1} * R'_{c_2} * ... * R'_{c_m} \\ Push[T_{R,s}] &= T_{R_{c_1},j_1} * T_{R_{c_2},j_2} * ... * T_{R_{c_m},j_m} \end{aligned} \tag{12}$$

Define

$$\begin{aligned} &\rho_1 = |R'_{c_1}| - |T_{R_{c_1},j_1}|, \\ &\rho_2 = |R'_{c_2}| - |T_{R_{c_2},j_2}|, \\ &\vdots \\ &\rho_m = |R'_{c_m}| - |T_{R_{c_m},j_m}| \end{aligned} \tag{13}$$

By the induction hypothesis, since $R'_{c_1} \subset R_{c_1}$, so $\rho_1 \geq 0$. And if $R'_{c_1}(r_N) \not\subset T_{R_{c_1},j_1}(r_N)$, $\rho_1 \geq 1$. This result holds for all the rest $\rho_2, \ldots, \rho_m$. Due to $|R * S| = |R| + |S|$ for any routing maps R and S, we have

$$\begin{aligned}\Delta &= |R'| - |Push[R']| \\ &+ |R'_{c_1}| + |R'_{c_2}| + ... + |R'_{c_m}| \\ &- |T_{R_{c_1},j_1}| - |T_{R_{c_2},j_2}| - ... - |T_{R_{c_m},j_m}| \\ &+ |Push[T_{R,s}]| - |T_{R,s}| \\ &= |R'| - |Push[R']| + \rho_1 + \rho_2 + ... + \rho_m \\ &+ |Push[T_{R,s}]| - |T_{R,s}| \end{aligned}$$

We have define $|R|$ as the number of occupied nodes in the tree, if R is on P_N and $N \geq 2$, then $|R|$ equals the number occupied nodes among the root and root's children nodes in R (denote by N_1), plus the number of occupied nodes among the rest low level nodes of R (denote by N_2), so $|R| = N_1 + N_2$. *From now on, there is a very important remark on notation $|R|$: for brevity, we omit the N_2 operand and only write $|R| = N_1$.* The reason of this simplification is, in the following proof we only care about the occupation of a tree's root and root's children nodes, letting $|R| = N_1$ will help to understand this occupation status, and will not affect the correctness of the proof.

Again, there are three cases to consider:

Case 1. $|R'| - |Push[R']| = 1$. $\Delta = 1 + \rho_1 + \rho_2, ..., \rho_m + |Push[T_{R,s}]| - |T_{R,s}| \geq 1$, it's done.

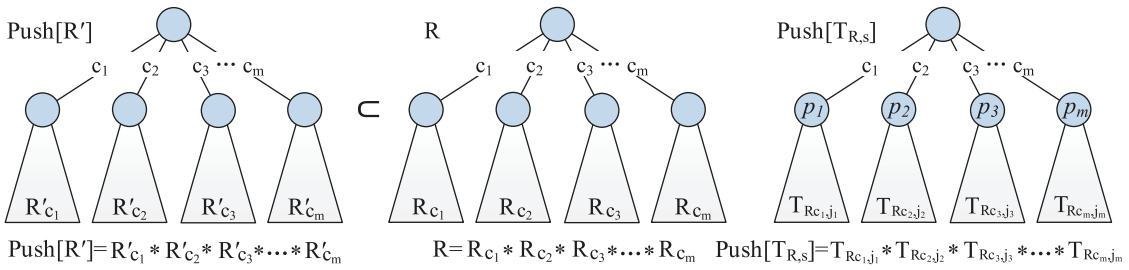


Fig. A.14. Induction illustration for routing maps: $R' \subset R \Rightarrow R'_{c_1} \subset R_{c_1}, R'_{c_2} \subset R_{c_2}, ..., R'_{c_m} \subset R_{c_m}$.

Case 2. $|R'| - |Push[R']| = 0$. Hence $\Delta = \rho_1 + \rho_2, ..., \rho_m + |Push[T_{R,s}]| - |T_{R,s}| \geq 0$. Since $|T_{R,s}| \in [1, m]$[5], we have two subcases to consider:

Subcase 1: $|T_{R,s}| = n \in [1, m-1]$, then $|Push[T_{R,s}]| - |T_{R,s}| \geq 1$, so $\Delta \geq 1$, it's done.

Subcase 2: $|T_{R,s}| = m$, then $\Delta = \rho_1 + \rho_2, ..., \rho_m \geq 0$. Remember that $p_1 = T_{R_{c_1}, j_1}(r_N), p_2 = T_{R_{c_2}, j_2}(r_N), ..., p_m = T_{R_{cm}, j_m}(r_N)$. In this case, all the next hops in $p_1, p_2, \ldots, p_m$ have a population of 1, which means $p_1 \cap p_2 \cap ... \cap p_m = \emptyset$, so in Step 2 $\texttt{PrevalentSelect}(p_1, p_2, ..., p_m) = p_1 \cup p_2 \cup ... \cup p_m = M_R$. Because here $|Push[R']| = m$, and $|R'| - |Push[R']| = 0$, so $|R'| = m$. Hence there are two scenarios for G to consider:

(a) $G = \emptyset$, then $G \subset M_R$, it's done.
(b) $G \neq \emptyset$, then $Push[R']$ pushes G to exactly one of the

children vertices of the root node in R', i.e., one from $R'_{c_1}(r_N), R'_{c_2}(r_N), ..., R'_{cm}(r_N)$. Let's assume it is $R'_{c_i}(r_N)$, so $G \subset R'_{c_i}(r_N)$. Two scenarios to consider: 1) $R'_{c_i}(r_N) \subset p_i$, then $G \subset M_R$, it's done. 2) $R'_{c_i}(r_N) \not\subset p_i$, then $\rho_i \geq 1$, so $\Delta \geq 1$, and it's done.

Case 3. $|R'| - |Push[R']| \in [-(m-1), -1]$. Since $|Push[R']| = m$, then $|R'| \in [1, m-1]$ ($|R'| = m$ has been considered in Case 2). Again, there are two subcases for $|R'|$ to consider.

Subcase 1: $|R'| = 1$. In this case $G \neq \emptyset$ and all the m children vertices of the root in R' are empty, so we have $G \subset R'_{c_1}(r_N), G \subset R'_{c_2}(r_N), ..., G \subset R'_{cm}(r_N)$. Two situations to consider:

(1) If $R'_{c_1}(r_N) \subset p_1, R'_{c_2}(r_N) \subset p_2, ..., R'_{cm}(r_N) \subset p_m$, then $\rho_1 \geq 0, \rho_2 \geq 0, \ldots, \rho_m \geq 0$ and $p_1 \cap p_2 \cap ... \cap p_m \neq \emptyset$. Hence $M_R = \texttt{PrevalentSelect}(p_1, p_2, ..., p_m) = p_1 \cap p_2 \cap ... \cap p_m$, which yields $G \subset M_R$ and $|T_{R,s}| = 1$. Then $\Delta = 1 - m + \rho_1 + \rho_2 + ... + \rho_m + m - 1 \geq 0$, it's done.
(2) Only t ($t < m$) of $R'_{c_1}(r_N), R'_{c_2}(r_N), ..., R'_{cm}(r_N)$ is a subset of $p_1, p_2, \ldots, p_2$. Assume that it is the first t children node of the root in R', then $R'_{c_1}(r_N) \subset p_1, R'_{c_2}(r_N) \subset p_2, ..., R'_{c_t}(r_N) \subset p_t, R'_{c_{t+1}}(r_N) \not\subset p_{t+1}, ..., R'_{cm}(r_N) \not\subset p_m$, which means $G \subset p_1, G \subset p_2, ..., G \subset p_t, G \not\subset p_{t+1}, ..., G \not\subset p_m$, and $\rho_1 \geq 0, \rho_2 \geq 0, ..., \rho_t \geq 0, \rho_{t+1} \geq 1, ..., \rho_m \geq 1$.

Three scenarios about G to consider:

(a) $G \not\subset M_R$: we need to prove $\Delta \geq 1$. Assume $u = \texttt{RandSelect}(M_R)$ and q children vertices will be filled with u in $Push[T_{R,s}]$. In this case $q > t$, so $q - t \geq 1$, and $|T_{R,s}| = m + 1 - q$, yielding $|Push[T_{R,s}]| - |T_{R,s}| = m - (m + 1 - q) = q - 1$. While $|R'| - |Push[R']| = 1 - m$, therefore $\Delta = 1 - m + \rho_1 + \rho_2 + ... + \rho_m + (q-1) = q - m + \rho_1 + \rho_2 + ... + \rho_m = q - m + \rho_1 + \rho_2 + ... + \rho_t + (\rho_{t+1} - 1) + ... + (\rho_m - 1) + (m - t) = q - t + \rho_1 + \rho_2 + ... + \rho_t + (\rho_{t+1} - 1) + ... + (\rho_m - 1) \geq 1$, it's done.
(b) $G \subset M_R$ but $G \neq \texttt{RandSelect}(M_R)$: we need to prove $\Delta \geq 0$. In this case $q = t$, and $\Delta = q - t + \rho_1 + \rho_2 + ... + \rho_t + (\rho_{t+1} - 1) + ... + (\rho_m - 1) \geq 0$.
(c) $G = \texttt{RandSelect}(M_R)$: we need to prove $\Delta \geq 0$. In this case $q = t$, and $\Delta = q - t + \rho_1 + \rho_2 + ... + \rho_t + (\rho_{t+1} - 1) + ... + (\rho_m - 1) \geq 0$, it's done.

Subcase 2: $|R'| \in [2, m-1]$. In this case $G \neq \emptyset$ and at least 2 children vertices of the root in R' are empty. Denote by s ($s \in [2, m-1]$) the number of empty children vertices of the root in R'. Then $Push[R']$ assigns G to the s children vertices of the root in $Push[R']$, so $|R'| = 1 + m - s$. Assume that first t children of the root node in R' is a subset of $p_1, p_2, \ldots, p_m$, namely $R'_{c_1}(r_N) \subset p_1, R'_{c_2}(r_N) \subset p_2, ..., R'_{c_t}(r_N) \subset p_t, R'_{c_{t+1}}(r_N) \not\subset p_{t+1}, ..., R'_{cm}(r_N) \not\subset p_m$, so $\rho_1 \geq 0, \rho_2 \geq 0, ..., \rho_t \geq 0, \rho_{t+1} \geq 1, ..., \rho_m \geq 1$. Assume that the next hop(s) in M_R selected by $\texttt{PrevalentSelect}(p_1, p_2, ..., p_m)$ has a population of q, now $|T_{R,s}| = 1 + m - q$, $\Delta = (1 + m - s) - m + \rho_1 + \rho_2 + ... + \rho_m + m - (m - q + 1) = 1 - s + \rho_1 + \rho_2 + ... + \rho_t + (\rho_{t+1} - 1) + ... + (\rho_m - 1) + m - t + q - 1 = m - s + q - t + \rho_1 + \rho_2 + ... + \rho_t + (\rho_{t+1} - 1) + ... + (\rho_m - 1)$. Apparently $m - s \geq 1$ (because $s \in [2, m-1]$). In the following, three scenarios for G need to be considered:

(a) $G \not\subset M_R$: we need to prove $\Delta \geq 1$. In this case $q > t$, so $q - t \geq 1$, therefore $\Delta \geq 2$, it's done.
(b) $G \subset M_R$ but $G \neq \texttt{RandSelect}(M_R)$: we need to prove $\Delta \geq 0$. In this case $q = t$, so $\Delta \geq 1$, it's done.
(c) $G = \texttt{RandSelect}(M_R)$: we need to prove $\Delta \geq 0$. In this case still $q = t$, so $\Delta \geq 1$, it's done!
The proof for the component trie completes.

For the character trie, since Pass one complements all the children nodes of characters that are absent, let $m = 26$ and our theorem holds. For the binary trie, the ORTC algorithm has been proven [30].

[5] $|T_{R,s}|$ should be actually in range $[1 + N_2, m + N_2]$. Here we apply the simplification on $|R|$ for the first time, so $|T_{R,s}|$ is redirected into range $[1, m]$.

References

[1] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, A. Yadav, A global name service for a highly mobile internet, in: Proceedings of ACM SIGCOMM'14, 2014.

[2] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K.H. Kim, S. Shenker, I. Stoica, A data-oriented (and beyond) network architecture, in: ACM SIGCOMM, ACM, 2007, pp. 181–192.

[3] J. Rajahalme, M. Särelä, K. Visala, J. Riihijärvi, On name-based inter-domain routing, Comput. Netw. 55 (4) (2011) 975–986.

[4] D.G. Thaler, C.V. Ravishankar, Using name-based mappings to increase hit rates, IEEE/ACM Trans. Netw. (TON) 6 (1) (1998) 1–14.

[5] H. Hwang, S. Ata, M. Murata, Frequency-aware reconstruction of forwarding tables in name-based routing, in: Proceedings of the 5th International Conference on Future Internet Technologies, ACM, 2010, pp. 45–50.

[6] M. Gritter, D.R. Cheriton, An architecture for content routing support in the internet, in: Proceedings of the USITS, 2001, pp. 4–15.

[7] M. Caesar, M. Castro, E.B. Nightingale, G. O'Shea, A. Rowstron, Virtual ring routing: network routing inspired by DHTs, in: Proceedings of ACM SIGCOMM, ACM, New York, NY, USA, 2006, pp. 351–362.

[8] A. Carzaniga, M.J. Rutherford, A.L. Wolf, A routing scheme for content-based networking, in: Proceedings of IEEE INFOCOM, 2004, pp. 918–928.

[9] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K.H. Kim, S. Shenker, I. Stoica, A data-oriented (and beyond) network architecture, in: Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07), in: SIGCOMM '07, ACM, New York, NY, USA, 2007, pp. 181–192, doi:10.1145/1282380.1282402.

[10] H. Hwang, S. Ata, M. Murata, A feasibility evaluation on name-based routing, in: IP Operations and Management, in: Lecture Notes in Computer Science, 5843, Springer Berlin Heidelberg, 2009, pp. 130–142, doi:10.1007/978-3-642-04968-2_11.

[11] A. Singla, P.B. Godfrey, K. Fall, G. Iannaccone, S. Ratnasamy, Scalable routing on flat names, in: Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT'10), in: CoNEXT'10, ACM, New York, NY, USA, 2010, pp. 20:1–20:12, doi:10.1145/1921168.1921195.

[12] A. Detti, N. Blefari-Melazzi, S. Salsano, M. Pomposini, CONET: A content centric inter-networking architecture, in: Proc. of ACM ICN'11, 2011.

[13] S. Jain, Y. Chen, Z.-L. Zhang, Viro: a scalable, robust and namespace independent virtual id routing for future networks, in: Proceedings IEEE INFOCOM, 2011, pp. 2381–2389, doi:10.1109/INFCOM.2011.5935058.

[14] K. Xu, H. Zhang, M. Song, J. Song, A content aware and name based routing network speed up system, in: Proceedings of the International Conference on Pervasive Computing and the Networked World, in: ICPCA/SWS'12, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 672–688.

[15] K.V. Katsaros, N. Fotiou, X. Vasilakos, C.N. Ververidis, C. Tsilopoulos, G. Xylomenos, G.C. Polyzos, On inter-domain name resolution for information-centric networks, in: Proceedings of the International IFIP TC 6 Conference on Networking - Volume Part I, in: IFIP'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 13–26, doi:10.1007/978-3-642-30045-5_2.

[16] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, Q. Dong, Namefilter: achieving fast name lookup with low memory cost via applying two-stage bloom filters, in: Proceedings of IEEE INFOCOM'13, 2013.

[17] L. Zhang, D. Estrin, V. Jacobson, B. Zhang, Named Data Networking (NDN) Project, in: Technical Report, NDN-0001, 2010.

[18] V. Jacobson, D.K. Smetters, J.D. Thornton, M. Plass, N. Briggs, R. Braynard, Networking named content, in: Proceedings of CoNEXT, 2009.

[19] A.V. Vasilakos, Z. Li, G. Simon, W. You, Information centric network: research challenges and opportunities, J. Netw. Comput. Appl. 52 (0) (2015) 1–10.

[20] J. Garcia-Luna-Aceves, Name-based content routing in information centric networks using distance information, in: Proceedings of the 1st international conference on Information-centric networking, ACM, 2014, pp. 7–16.

[21] R. Ahmed, M. Bari, S.R. Chowdhury, M. Rabbani, R. Boutaba, B. Mathieu, et al., αroute: a name based routing scheme for information centric networks, in: Proceedings IEEE INFOCOM'13, IEEE, 2013, pp. 90–94.

[22] E. Baccelli, C. Mehlis, O. Hahm, T.C. Schmidt, M. Wählisch, Information centric networking in the IoT: Experiments with NDN in the wild, in: Proceedings of the 1st International Conference on Information-centric Networking, in: INC'14, 2014, pp. 77–86.

[23] C. Tsilopoulos, G. Xylomenos, Y. Thomas, Reducing forwarding state in content-centric networks with semi-stateless forwarding, in: INFOCOM, 2014 Proceedings IEEE, IEEE, 2014, pp. 2067–2075.

[24] A. Hoque, S.O. Amin, A. Alyyan, B. Zhang, L. Zhang, L. Wang, NLSR: named-data link state routing protocol, in: Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking, ACM, 2013, pp. 15–20.

[25] L. Wang, A. Hoque, C. Yi, A. Alyyan, B. Zhang, OSPFN: an OSPF based routing protocol for named data networking, University of Memphis and University of Arizona, Tech. Rep (2012).

[26] W. ying Ma, B. Shen, J. Brassil, Content services network: the architecture and protocols, in: Proceedings of the 6th IWCW, 2001, pp. 89–107.

[27] W. Yi, H. Keqiang, D. Huichen, M. Wei, J. Junchen, L. Bin, C. Yan, Scalable name lookup in ndn using effective name component encoding, in: Proceedings of IEEE ICDCS'12, 2012.

[28] H. Dai, B. Liu, Y. Chen, Y. Wang, On pending interest table in named data networking, in: Proceedings of ACM/IEEE ANCS'12, 2012, pp. 211–222.

[29] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, D. Yang, Wire speed name lookup: a gpu-based approach, in: Proceedings of USENIX NSDI'13, 2013.

[30] R.P. Draves, C. King, S. Venkatachary, B.D. Zill, Constructing optimal IP routing tables, in: Proceedings of IEEE INFOCOM'99, 1999, pp. 88–97.

[31] E. Fredkin, Trie memory, Commun. ACM 3 (9) (1960) 490–499.

[32] The openmp api specification for parallel programming, Http://openmp.org/wp/.

[33] V. Srinivasan, G. Varghese, Faster ip lookups using controlled prefix expansion, in: Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, in: SIGMETRICS '98/PERFORMANCE '98, 1998, pp. 1–10.

[34] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, in: Proceedings of the ACM SIGCOMM'97, 1997, pp. 3–14.

[35] S. Nilsson, G. Karlsson, Ip-address lookup using lc-tries, IEEE J. Select. Areas Commun. 17 (6) (1999) 1083–1092.

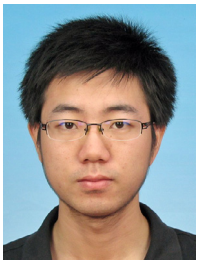
[36] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software ip lookups with incremental updates, ACM SIGCOMM Comput. Commun. Rev. 34 (2) (2004) 97–122.

[37] M. Bando, H.J. Chao, Flashtrie: hash-based prefix-compressed trie for ip route lookup beyond 100gbps, in: Proceedings of IEEE INFOCOM'10, IEEE, 2010, pp. 1–9.

[38] M.J. Atallah, Algorithms and theory of computation handbook, 2nd Edition, Addison-Wesley.

[39] A. Andersson, S. Nilsson, Improved behaviour of tries by adaptive branching, Inf. Process. Lett. 46 (6) (1993) 295–300.

[40] H. Dai, J. Lu, Y. Wang, B. Liu, BFAST: Unified and scalable index for ndn forwarding architecture, in: Proceedings of the IEEE INFOCOM, 2015, pp. 2290–2298.



Huichen Dai: is a postdoctoral research fellow in the Department of Computer Science and Technology, Tsinghua University. He got his B.S. degree from Xi'an University of Electronic Science and Technology, Xi'an, China, in 2010. His research interests mainly lie in: router architecture, fast packet processing, future Internet Architecture, e.g., Named Data Networking (NDN), Software Defined Network (SDN). He now serves as the reviewer for IEEE Communications Letters and IEEE Communications Magazine. http://s-router.cs.tsinghua.edu.cn/~daihuichen/

Bin Liu: has been a full professor in the Department of Computer Science and Technology, Tsinghua University, China since 1999. Bin Liu received his B.S., M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1985, 1988 and 1993, respectively. From 1993 to 1995, he was a Postdoctoral Research Fellow in the National Key Laboratory of SPC and Switching Technologies, Beijing University of Post and Telecommunications, Beijing, China. In 1995, he transferred to the Department of Computer Science and Technology, Tsinghua University. http://s-router.cs.tsinghua.edu.cn/~liubin/index.htm