Scalable Name Lookup with Adaptive Prefix Bloom Filter for Named Data Networking

Wei Quan, Changqiao Xu, Jianfeng Guan, Hongke Zhang, and Luigi Alfredo Grieco

***Abstract*—In Named Data Networking (NDN), packet forwarding decisions rely upon lookup operations on variable-length hierarchical names instead of fixed-length host addresses. This pivotal feature introduces new challenges in the deployment of NDN at the Internet scale. In this letter, a novel Name Lookup engine with Adaptive Prefix Bloom filter (NLAPB) is conceived, in which each NDN name/prefix is split into B-prefix followed by T-suffix. B-prefix is matched by Bloom filters whereas T-suffix is processed by the small-scale trie. The length of B-prefixes (and T-suffixes) is dynamically throttled based on their popularity in order to accelerate the lookup. Experimental results show that: (i) NLAPB is able to lower the false positive rate with respect to a lookup entirely based on Bloom filters; (ii) it decreases the memory requirements with respect to a trie-based approach; (iii) it reduces processing time with respect to both them.**

***Index Terms*—Scalable Name Lookup; Adaptive Prefix; NDN.**

I. INTRODUCTION

NAMED Data Networking (NDN) [1] is a promising Information Centric Networking (ICN) architecture, in which packets forwarding decisions are driven by content names instead of IP addresses. In NDN, most lookup operations rely upon the longest prefix matching (LPM) of content names (see Fig. 1). Unlike fixed-length IP addresses, NDN names with *variable lengths* have a *hierarchical structure* consisting of a sequence of *delimited components*, which may contain any kind of characters. These emerging features bring unprecedented challenges for existing LPM solutions in terms of both efficiency and scalability, and make name lookup in NDN extremely challenging in a practical large-scale use.

A basic LPM solution for IP address is the *trie* [2], an ordered tree data structure, based on which a family of related solutions have been conceived. Recently, Wang *et al.* investigated its usage in NDN and proposed an effective name component encoding for name prefix trie to accelerate name lookup [3]. Unfortunately, it inherits the well-known memory efficiency issues of all trie implementations, which become very relevant in the presence of deep NDN name trees.

Manuscript received xxx xx, 2013. The associate editor coordinating the review of this letter and approving it for publication was xxx.

W. Quan, C. Xu, J. Guan and H. Zhang are with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecom., Beijing, China (e-mail: quanwei, cqxu, jfguan, hkzhang@bupt.edu.cn).

L. A. Grieco is with the Department of Electrical and Information Engineering, Politecnico di Bari, Bari, Italy (e-mail: a.grieco@poliba.it).

This work was supported by the National Basic Research Program of China (973 Program) under Grant No. 2013CB329102, the National Natural Science Foundation of China (NSFC) under Grant No. 61372112 and 61232017, and the PON project RES NOVAE funded by the Italian MIUR and by the European Union (European Social Fund).

Digital Object Identifier xxxxxx

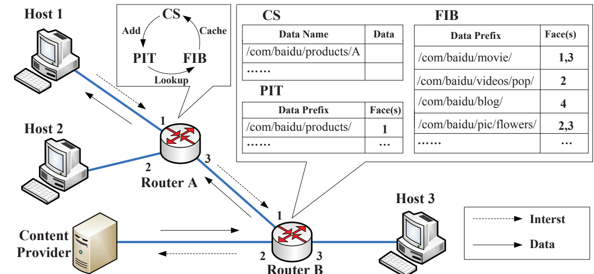

Fig. 1. NDN name-based forwarding diagram.

Another family of alternative approaches rely on *Bloom filters* (BF) [4]. Since standard BF does not allow element deletions, *Counting* BF (CBF) is usually adopted in related applications [5]. Sarang *et al.* first applied the CBF in the LPM for IP addresses [6]. Recently, Wang *et al.* proposed an efficient lookup scheme for NDN by applying two-stage CBFs [7]. Won *et al.* also proposed a NDN software forwarding lookup engine based on hash tables combined with BF [8].

Due to hash collisions, CBF efficiency is negatively affected by the *false positive rate* (*fpr*) that is determined inherently by the number of inserted entries. On the other hand, each hash computing is time-consuming, which is dependent on the length of the entry. As a consequence, because of the huge size of long NDN names, solutions entirely based on CBF would be severely impaired in NDN. In addition, if we consider the two-stage approach in [7], we can discover that BFs are adopted in two continuous stages, which will amplify the *fpr*.

This letter investigates the characteristics of NDN names and conceives a NDN-customized Name Lookup solution with Adaptive Prefix Bloom filter (NLAPB). *NLAPB is the first solution in literature to split NDN prefixes into two segments and conduct the lookup operations with CBF and trie for each of them respectively. With this novel hybrid data structure, NLAPB can alleviate all aforementioned drawbacks of trie- and CBF-based solutions as well as capitalize their strengths.*

II. PROBLEM FORMULATION AND MOTIVATION

Problem Formulation: Let $\mathcal{U}$ be the set of NDN names, and $\mathcal{S}$ be the set of name prefixes. Given any name $q \in \mathcal{U}$, we can determine the longest matching prefix x_q in $\mathcal{S}$ for q with an efficient name lookup engine $\mathcal{L}$. In a more formal way, this problem can be expressed as follows:

$$\begin{aligned} \min_{\mathcal{U},\mathcal{S},\mathcal{L}} \quad & \mathcal{T} = \mathbb{T}\left(q \in \mathcal{U}, \mathcal{S}, \mathcal{L}\right) \\ \text{subject to} \quad & |\mathcal{U}| >> 1, |\mathcal{S}| >> 1, \\ & \mathcal{M} \leqslant \varepsilon, \\ & fpr \leqslant \sigma. \end{aligned} \tag{1}$$

where $\mathbb{T}(\cdot)$ is the function of lookup time, ε is the maximal available memory, and σ is a threshold of the *fpr*.

Our aim is to reduce the lookup time $\mathcal{T}$ as much as possible with a reasonable memory cost $\mathcal{M}$ and a small *fpr* to support large scale scenarios.

Motivations of our solution: To solve the problem (1), we focus on a novel hybrid lookup engine that jointly adopts CBF and trie. It is motivated by the following practical considerations:

(1) NDN names consist of a sequence of delimited components, which makes it difficult to determine how many filters are needed when pure CBF-based solutions are adopted.

(2) Long NDN names can be split into two shorter segments: the first one (B-prefix) having a fixed length and the second one (T-suffix) having a variable length. They can be separately treated using CBF and trie-based approaches, respectively.

(3) The hierarchical NDN names enable aggregation, which implies the number of different B-prefixes will be limited in a small range if a suitable length is adopted. In this case, it becomes possible the adoption of CBF with a small *fpr*.

(4) The trie data structure has a high flexibility and scalability by dynamic memory allocation, which enables to process the T-suffixes with variable lengths efficiently.

III. Design of Proposed Solution: NLAPB

A. Key concepts

We present a novel hybrid data structure made of CBF and trie. In this proposal, each name/prefix is devided into two parts: *B-prefix* and *T-suffix*. Fig. 2 illustrates the diagram of this structure. B-prefixes, $B_1, B_2 \cdots$, with a limited length, can be efficiently stored and processed using the CBFs; and T-suffixes, $T_1, T_2 \cdots$, instead, with variable lengths which are shorter than their full names, can be handled easily using trie, thus constructing a set of small-scale T-suffix trees which are bind to the B-prefixes with a hash table.

More detailed, one filter is allocated for all B-prefixes of each length, and the size of each filter is dimensioned by its share of the total number of B-prefixes with different lengths. This configuration will avoid the asymmetric behaviors among different CBFs. When a prefix needs to be inserted or removed, its B-prefix is first added to or deleted from the filter, thus, the counters corresponding to the hash values are incremented or decremented. Then the hash table connecting B-prefixes with T-suffixes trees is updated. In the T-suffixes tree, each encoding component is corresponding with one element of trie. The starting element (root) of a trie is bound to a B-prefix with the hash table. In Fig. 2, the parents of root elements are

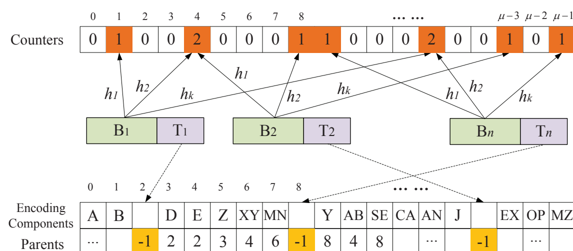

Fig. 2. Diagram of novel data structure.

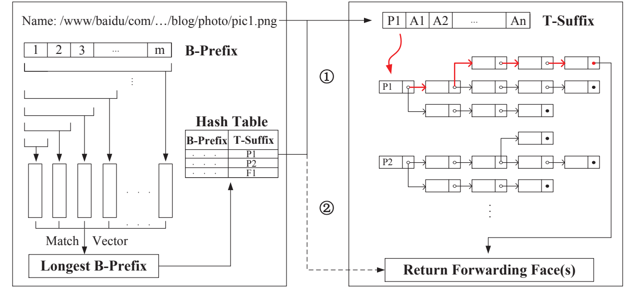

Fig. 3. Name prefix lookup framework.

labeled as “-1”. With the knowledge of the root, the following components can be inserted or removed along the trie by dynamic memory allocation and recycling.

The proposed lookup process mainly includes two stages based on the separation of B-prefix and T-suffix, which are totally different from the NameFilter in [7]. Fig. 3 shows this name lookup framework. In the first stage, the B-prefix of a name is first processed by CBFs, and a hash table assists to ascertain the location of corresponding T-suffix tree. Of course, if the name/prefix is short enough, the outgoing face(s) will be returned directly. In the second stage, with the knowledge of the root information (*e.g.* P1), the prefix matching continues based on the trie structure until a longest matching prefix is searched. Then, the corresponding forwarding face(s) will be returned. Similarly, the updates also follow this two-stage process. Note that there is a big probability that many different content names/prefixes share the same B-prefix, thus, the number of different B-prefixes will be small, and the possibility of updating in CBFs will be reduced.

To sum up, the hybrid data structure we propose herein can improve lookup operations in NDN thanks to the following key features: *(i) it reduces the number of entries inserted into CBFs, thus lowering the false positive rate; (ii) it shortens the variable part of the NDN name thanks to the adoption of T-suffixes, which can be effectively handled using a trie; (iii) it bounds the number of CBFs to adopt in the lookup engine, thus easing their hardware implementation.* In this way, it becomes possible to profit from the advantages of both CBF- and trie-based techniques while counteracting their main limitations.

B. Setting boundaries in NDN names

An essential task for us is how two set a suitable boundary between B-prefix and T-suffix in order to solve the problem (1). To this end, we collected about 40 million URLs from the Internet to build a solid dataset. In order to keep consistent with the format of NDN names as much as possible, we preprocessed the URLs such as adding the *version* and *segment* information, which is strictly in accordance with [9].

We focus on the statistical characteristics of collected names in two terms: *the distribution of the total number of components in each name* and *the differentiation ratio of prefixes with different levels*. The first parameter indicates the length of name (expressed in number of components separated by “/”) which helps to tune the proper boundary that maximizes lookup efficiency. The second one, instead, expresses the degree of aggregation in the name space, which is tightly linked to the *fpr* and the space cost of the lookup algorithm.

We calculate the total number of prefixes with i components, Σ_i, and count all different prefixes, Λ_i, among them. Then, we define the aggregation ratio $d_i = 1 - \Lambda_i/\Sigma_i$, while the differentiation ratio is $1 - d_i$.

Through the parameter estimation and distribution fitting, we observe some essential statistical findings. In Fig. 4(a), the length of prefixes follows a Gamma-distribution, $F \sim \mathrm{Gamma}(\lambda, \theta)$. Its parameters are: shape $\lambda = 16.23$, scale $\theta = 0.31$, mean 5.14, and variance 1.63. Thus, the probability density function using the shape-scale parameterization is

$$f(z; \lambda, \theta) = \frac{1}{\theta^{\lambda}} \frac{1}{\Gamma(\lambda)} z^{\lambda-1} e^{-\frac{z}{\theta}} \quad (2)$$

where $z, \lambda, \theta > 0$ and $\Gamma(\lambda)$ is the gamma function with λ.

The probability of names with a length smaller than t is

$$P(z \leqslant t) = \int_0^t f(z; \lambda, \theta) dz = \frac{\gamma(\lambda, t/\theta)}{\Gamma(\lambda)} \quad (3)$$

where $\gamma(\lambda, t/\theta)$ is the lower incomplete gamma function.

It implies that a proportion of $\frac{\gamma(\lambda, t/\theta)}{\Gamma(\lambda)}$ of names will be processed only in the first stage using CBF, if the boundary is set as t.

Further, we can get the mean length of T-suffixes as

$$\begin{aligned} l^T &= \int_t^{\infty} (z - t) f(z; \lambda, \theta)\, dz \\ &= \int_t^{\infty} z f(z; \lambda, \theta)\, dz - t \int_t^{\infty} f(z; \lambda, \theta)\, dz = \lambda\theta - \delta \end{aligned} \quad (4)$$

where $\delta > 0$ is a value dependent on t.

Hence, the space complexity in T-suffixes matching reduces to $O((\lambda\theta - \delta) \cdot \Omega)$, where Ω is the total number of name items, and time complexity achieves to $O(\lambda\theta - \delta)$ in the average case.

Besides, it is noted that Fig. 4(a) shows an asymmetric shape, where there are different number of samples over the set of prefixes lengths. This implies that the memory should be proportionally allocated and tuned to each CBF based on its share of the total number of prefixes to minimize the *fpr*, hence achieving the desired performance.

In Fig. 4(b), we observe that the differentiation ratio is very low for prefixes of a few components, and increases as the number of components increases. This implies that the number of prefixes inserted into each CBF should be estimated if the total number of prefixes is known.

According to [6], on the condition of an optimal number of hash functions $k = \ln 2 \cdot \hbar/\mu$, we can get the *fpr* satisfying:

$$\ln fpr = -(\ln 2)^2 \cdot \hbar \big/ \mu. \quad (5)$$

where $\hbar$ denotes the size of filter, μ is the cardinality of the set of name/prefixes that can be inserted into the filter.

If we consider two sets of names with different cardinalities μ_0 and μ_1, and define $\omega = \mu_0/\mu_1$. Based on eq. (5), we can easily get a relation:

$$fpr_1 = {fpr_0}^{\omega} \quad (6)$$

Eq. (6) tells that fpr follows an exponential law with ω. In fact, the bigger the ω is, the smaller the *fpr* is (if using CBFs).

Combining the observations from Fig. 4, we finally get an empirical suitable value for the boundary threshold t, which is suggested $t = 5$. Based on the preceding analysis, there are two main reasons. Firstly, this threshold allows a considerable amount of names/prefixes to be processed using CBFs efficiently, obviously shortens T-suffix to be precessed using trie, and reduces the memory cost greatly. Secondly, it ensures a small differentiation ratio of prefixes, which limits the *fpr* and the updating frequency of the CBF.

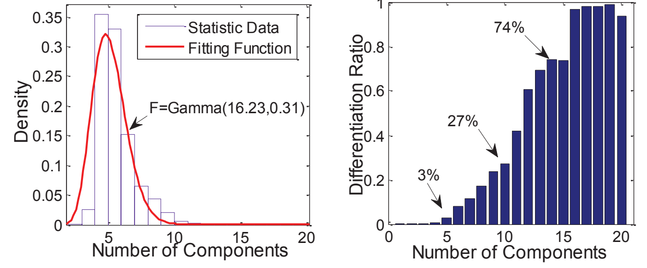

Fig. 4. The statistical results (a) the distribution of the number of name components (b) the differentiation ratio of prefixes.

C. Adaptive prefix bloom filter based on popularity

Now a key ingredient is still missing for the lookup process. With a certain value t, every time the router lookups popular prefixes, the matching process conducts many repetitive memory accesses for T-suffixes in the second stage. To reduce the unnecessary repeating operations, *the length of B-prefix should vary with popularity of a prefix: the higher the popularity is, the longer the B-prefix should be*. To this end, we further propose an adaptive mechanism based on our proposed framework to adjust B-prefix length in a certain range adaptively.

In this sheme, a counting container is adopted to record lookup times, $C_i \in \mathbb{N}^+$, for the ith B-prefix in a certain interval. According to the Zipf law, each B-prefix will own a popularity level, $r \in \{1, 2, \cdots m\}$, which is related with its lookup times C_i. Then, a mapping rule is adopted to map the popularity level to a limited dynamic window, $r \rightarrow r'$, $r' \in [-b, b]$. To this end, when the popularity arises/descends to a new level r, an update process is triggered to lengthen/shorten B-prefix length by $|r'|$, hence adjusting the T-suffixes. Algorithm 1 shows this adaptation process in detail.

Algorithm 1: Adaptive Prefix Bloom Filter Mechanism

```
1:  procedure UpdateBFLength(Prefix)
2:      (C_i, r)←Lookup B-prefix(Prefix);
3:      r′ ← GetMapping(r);
4:      if r′ == 0 then
5:          Count(C_i)++;
6:      elseif r′ > 0 then
7:          Lengthen B-prefix by adding r′ components;
8:          Update(B-prefix, T-suffix) for Prefix;
9:      elseif r′ < 0 then
10:         Shorten B-prefix by cutting −r′ components;
11:         Update(B-prefix, T-suffix) for Prefix;
12:     end if
13: end procedure
```

In summary, this adaptation achieves an optimization in term of the lookup efficiency. By means of this simple scheme, it is optimal to assign a relative longer B-prefix and corresponding shorter T-suffix for popular prefixes. It can efficiently avoid a lot of redundancy matching repetitions specially for popular prefixes, hence, further accelerating the processing of the whole system.

IV. Experiments Evaluation

We finally choose three exsiting NDN LPM methods for comparison, Components-Trie (CT) [3], Bloom-Hash (BH) [6] and Hash Table (HT) [8]. Besides, we compare the proposed solution in case of no adaptive mechanism, simplified as NLPB, where the threshold t is set as 5 statically. These solutions are implemented and conducted in a commodity router equipped with 8 interfaces. The high-speed caching is also adopted to avoid frequently low-speed memory accesses. Table I details key experimental configurations.

TABLE I
Key Experimental Configurations

Param.	Hardware Configuration	Parameter	Value
CPU	Intel Xeon (R) 2.27GHz	Num. of Prefixes	1M-10M
Cores	6 Cores	Num. of CBFs	5,9
OS	Linux CentOS-5 x86_64	Num. of Hashes	3,4,5
RAM	DDR3 12GB, SRAM	Popularity Levels	5000
Thread	Single	Zipf Parameter	0.64-1.03
Faces	8 Forwarding faces	Size of CBFs	0-300Mb

During the experiments, we randomly select 1-10 million prefixes and insert them into the forwarding table in a uniform distribution. Each prefix is randomly associated with one forwarding face. The popularity of prefixes is ruled by the Zipf law with a variable parameter $\alpha = 0.64 \sim 1.03$ (a practical range in literature [10]). Based on this, we divide all prefixes into 5,000 levels, and build the request prefixes set (accounts for 80% of the total prefixes) and the update prefixes set (accounts for 20%) including 10% deletions and 10% insertions. Then we conduct the *prefix lookup* and *route update* and calculate the time consumption, memory cost and the *fpr*. Each experiment repeats 40 tests, and the average performance with 95% confidence interval (CI) is calculated.

Fig. 5(a) presents the comparison in terms of lookup processing rate. It is observed that the processing rate decreases with the scale of prefixes. BH and HT have low processing rate and CT has a better one. The reason is that BH and HT must conduct time-consuming hash computings and cope with a number of conflicts. The NLPB with a fixed t=5 outperforms above them since the novel data structure adopted can achieve the reduction of *fpr* and accelerate the processing rate by shortening the entries. Furthermore, the NLAPB speeds up the process thanks to the adaptive adjustment based on the popularity. In Fig. 5(b), we observe that BH and CT use the smallest and the biggest amount of memory, respectively. NLAPB exhibits the intermediate behaviors. Note that the adaptive mechanism does not bring much memory cost, since the memory reduced by lengthening B-prefixes achieves a near balance with the increased one by shortening them.

In Fig. 6(a), we observe that BH has a relative high *fpr* within a limited memory, and it increases as the number of hashes decreases. NLPB and NLAPB achieve a decrease by more than 3 orders of magnitude with respect to BH. The reason is that the high differentiation ratio means a relative large number of different prefixes with a longer length, which brings an extremely high *fpr* if using CBF. In our solutions, it can be avoided by using the trie. Note that the improvement is consistent with preceding analysis in Eq. (6). Fig. 6(b)

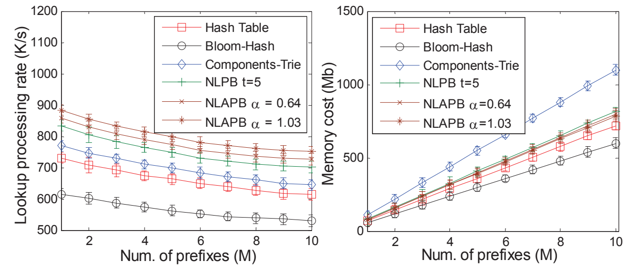

Fig. 5. (a) Lookup processing rate (b) Memory cost (95% CI).

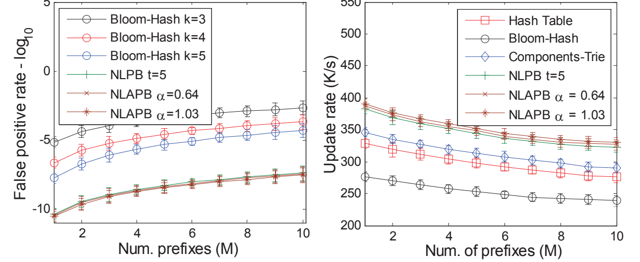

Fig. 6. (a) False positive rate (b) Update processing rate (95% CI).

shows the update processing rate. In this figure, NLPB and NLAPB achieve a superior performance than other solutions. The reason is that the separation of prefixes shortens the entries which reduces the frequency of updating CBFs and makes the processing more efficiently. In summary, these experiments illustrate NLAPB achieves a fairly guaranteed scalability (memory consumptions), low *fpr*, and high processing speeds (lookups and updates) when huge name spaces are considered.

V. Conclusion

This letter proposed a scalable name lookup solution, which builds a novel hybrid data structure made of CBF and trie, and can adjust the storage structure adaptively based on the prefix popularity. Experimental results verify its efficiency and scalability based on real-world URL names in terms of lookup and update efficiency, memory cost, and false positive rate. Our future work is to consider adopting more advanced hardware technologies such as GPU to further promote the name-based forwarding in practical large-scale use.

References

[1] V. Jacobson, D. K. Smetters, *et al.*, “Networking named content,” *Communications of the ACM*, vol. 55, pp. 117-124, 2012.
[2] E. Fredkin, “Trie memory,” *Commun. of the ACM*, vol. 3, no. 9, pp. 490-499, 1960.
[3] Y. Wang, K. He, *et al.*, “Scalable name lookup in NDN using effective name component encoding,” *IEEE ICDCS*, 2012.
[4] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. of the ACM*, vol. 13, no. 7, pp. 422-426, 1970.
[5] L. Fan, P. Cao, *et al.*, “Summary cache: A scalable wide-area web cache sharing protocol,” *ACM Trans. Netw.*, vol. 8, no. 3, pp. 281-293, 2000.
[6] S. Dharmapurikar, D. E. Taylor, *et al.*, “Longest prefix matching using Bloom filters,” *IEEE/ACM Trans. on Netw.*, vol. 14, pp. 397-409, 2006.
[7] Y. Wang, T. Pan, *et al.*, “NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters,” *IEEE INFOCOM mini-conference*, 2013.
[8] W. So, A. Narayanan, *et al.*, “Toward fast NDN software forwarding lookup engine based on Hash tables,” *ACM ANCS*, 2012.
[9] M. F. Bari, S. Chowdhury, *et al.*, “A survey of naming and routing in information-centric networks,” *IEEE Commun. Magazine*, vol. 50, no. 12, pp. 44-53, 2012.
[10] L. Saino, C. Cocora and G. Pavlou, “CCTCP: A scalable receiver-driven congestion control protocol for CCN,” *IEEE ICC*, 2013.