Reliable Memory Efficient Name Forwarding in Named Data Networking

Divya Saxena
Department of CSE
IIT Roorkee, India
divya.saxena.2015@ieee.org

Vaskar Raychoudhury
Department of CSE
IIT Roorkee, India
vaskar@ieee.org

Christian Becker
Chair for Information Systems II
Universität Mannheim, Germany
Ls-becker@uni-mannheim.de

Neeraj Suri
Dept. of Computer Science
TU Darmstadt, Germany
suri@cs.tu-darmstadt.de

***Abstract*—Named Data Networking (NDN) is a promising future Internet architecture which retrieves the content using their names. Content names composed of strings separated by '/' are stored in the NDN Forwarding Information Base (FIB) to forward the incoming packets further. To retrieve content through their names poses two main challenges for the NDN FIB: high memory consumption and high lookup time. Therefore, an efficient and scalable data structure is required to store names in FIB. Encoding components in all the names with a unique integer can reduce the memory consumption as well as lookup time. In this paper, we propose a scalable and memory-efficient radix trie based name component encoding scheme, *RaCE*, to implement NDN FIB. Our experiment results show that the *RaCE* scheme is reducing memory consumption by 89.95% and 26.07% compared to the original size of data and NCE scheme for the 29 million dataset, respectively.**

***Keywords*—Named Data Networking; NDN; Forwarding Information Base; FIB; Radix Trie; RaCE**

I. INTRODUCTION

Named Data Networking (NDN) is the future Internet architecture which has changed the semantics of communication from end-host content fetching to pulling the content from any intermediate node having valid content [1][2]. As a future Internet paradigm, NDNs network layer supports scalability (support to large Internet topology and high amount of name prefixes), security (integrity, origin authentication and relevance of routing information), resiliency (to detect and recover from packet delivery performance), and efficiency (support multi-path forwarding and in-network caching for efficient data dissemination).

In NDN, a content name (CN) is composed of several components of variable length which are maintained in the hierarchical structure, whereas current Internet uses fixed length (either 32 bit or 64 bit) addresses. In NDN, Forwarding Information Base (FIB) maintains name prefixes (NP) and their corresponding outgoing interface(s) and forwards incoming packets by calculating the longest prefix match (LPM) of the *CN*. To access contents via application name resolves numerous issues raised due to host-centric networking paradigm, but still NDN is facing two major issues. First is, FIB stores a huge number of names composed of an arbitrary number of components separated by '/'. Storing these huge number of variable length names in compared to IP address consumes high memory space. Second is, a lookup operation, such as insert, LPM, delete and update the *NPs* on the large number of names consumes high time.

To overcome the afore-mentioned issues, an efficient and scalable data structure is needed to implement NDN FIB. The data structure should have the following properties. First, it should consume less memory. Second, the operations like, insert, update, LPM, and delete should be performed in constant time. Third, LPM should not produce any false positives and false negatives. Therefore, the data structure should support low processing time, high throughput and reliability at the router level.

Some existing data structures, such as Character Component Trie (CCT), Name Prefix Trie (NPT), and Hash Table (HT) are used for exact-match algorithms. CCT and NPT both store some extra information, like each node of CCT stores a character, a Boolean value (whether the node is end of a name or not), number of child nodes and pointers to the child nodes. Moreover, HT is not memory-efficient and is not suitable for LPM.

A possible solution of the afore-mentioned problem is to encode the *name*, i.e., to encode the components of a *CN* with a unique integer where identical components share the same code. E.g., on an average, a *CN* can have 5-6 components and each component can have 6 characters. Then, the memory consumption will be approximately 30 bytes. To encode the component can solve this issue as it will consume only 20 bytes [3]. Wang, et al. [4] proposed the encoding for the NDN FIB. They encode the components using the integer on the current level states basis using the Encoded Name Prefix Trie (ENPT). The states and transitions of ENPT are maintained in the State Transition Array (STA). The use of array for implementing FIB can degrade the network and system performance as child nodes of the trie can be stored in consecutive memory locations. For each insertion and/or deletion operation in the STA, it is needed to re-allocate child nodes to other memory location. Sometimes, there can be need to re-allocate whole block of child nodes. For FIB implementation, it is impractical to assume that prior knowledge of content requests to arrive at an NDN router is

available before populating the data structure. Moreover, this may degrade system performance of the dynamic network environment.

We propose a memory-efficient Radix trie based Component encoding (*RaCE)* scheme, which supports components encoding of *CN* for NDN FIB implementation. *RaCE* encodes the name components (NC) in the 32-bit integer where identical component share the same integer. *RaCE* uses the Radix trie (*RT)* [5] as the basic structure which eliminates the impact of maintaining redundant information. *RT* also reduces the lookup time by compressing the search path. In summary, the contributions of this paper are listed below:

- We propose a component encoding scheme to implement NDN FIB.
- We use the Radix trie, a compact prefix trie to reduce the memory consumption and to support reliability at the router.
- Our extensive experiments over 29 million names dataset show that components encoding using *RaCE* is consuming 89.95% less memory than storing the names without encoding and reducing memory consumption by up to 26.07% in contrast to the NCE [4] scheme.

The paper organization is as follows. Section II discusses about the related research works. Section III presents a brief NDN introduction. Section IV describes our proposed scheme, its working, and the algorithm. Our performance results are given in Section V. Finally, Section VI concludes the paper and provides possible future directions.

II. RELATED WORK

In NDN, Wang, et al. proposed *Name Component Encoding* (NCE) [4] in which *NCs* are encoded using the integer and states are maintained in the STA. They extended this work to speed up the lookup time and proposed parallel architecture called *Parallel Name Lookup* (PNL) [6]. Some schemes [7][8] uses hash table (HT) with bloom filter (BF) (space-efficient probabilistic data structure) for supporting the fast name lookup in FIB. So, et al. [7] have presented an NDN forwarding lookup engine based on *HT* and *BF*. They have also proposed a collision resistant SipHash [8], for computing prefix hashes in one pass with simultaneous parsing of name components. *HT* consumes high memory while, *BFs* cannot be directly used for NDN's FIB.

Furthermore, Wang, et al. [11] proposed a two-stage Bloom filter-based scheme for finding LPM of *CN*. In first stage, *NPs* are stored into *BFs* and in second stage, *NPs* are divided into groups. This scheme performs better than character trie [9], NCE [4], BloomHash [10], and *BF*. Moreover, Wang, et al. [11] have shown two main technical issues to speed up name lookup of NDN FIB, such as, how to speed up the NPs' length calculation and how to speed up the *HT* operations. Later, Wang, et al. [12][13] proposed an adaptive greedy name lookup scheme to handle these issues. The above-mentioned schemes mainly based on the data structures, such as trie, *HT*, and *BF*. Trie based name lookup schemes having high lookup cost as their name prefix lookup time is based on the length of the name. *HTs* consumes high memory while, *BF* induces false positives [4]. To overcome these issues, Quan, et al. [14] proposed an Adaptive Prefix Bloom filter (NLAPB) scheme in which first part of a name matched by *BF*, while another part is processed by trie. Yuan, et al., [15] have proposed binary search of *HTs*. They have also proposed level pulling to reduce the number of *HT* lookup.

III. NDN OVERVIEW

In this section, we shall discuss about NDN working and FIB characteristics.

A. NDN Forwarding and Routing

Each NDN router uses three main data structures.

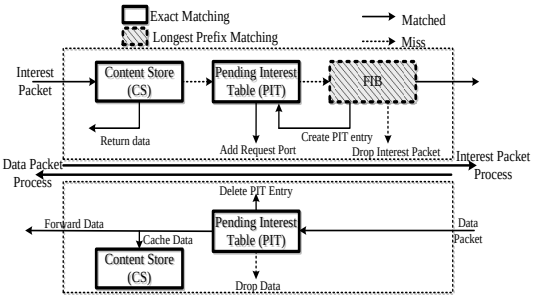

Fig.1. NDN forwarding [2]

- *Content Store (CS):* It temporarily stores actual data packets for future references.
- *Pending Interest Table (PIT):* It stores the requested packet's *CN* until corresponding data packet arrives.
- *Forwarding Information Base (FIB):* It forwards the incoming requesting packet to the next hop(s) using the LPM of *CN.*

NDN holds receiver-based communication, initiated by the content consumer. NDN communication uses two types of packets: *Interest packet (Ipkt)* and *Data packet (Dpkt)* (see Fig. 1). Each *Ipkt* and *Dpkt* contains *CN* for data to search. Whenever a content request reaches at the NDN router, requested data is searched using the *CN* associated with the *Ipkt*. If corresponding data is available at that router, a *Dpkt* is forwarded downstream using the trail of *Ipkt*. Otherwise, *Ipkt* is forwarded further. Therefore, the fundamental aim of the NDN is to access content without concerning about location of the content hosting entity. In NDN, security is in-built in the architecture. Every *Dpkt* in NDN is cryptographically signed by the content provider with its name for securing the data. Security layer provides security to each and every piece of content, unlike securing the entire communication channel in Internet.

IV. OUR PROPOSED SOLUTION

In this section, we discuss our proposed scheme for *NC* encoding using the Radix trie (RT). For the rest of this paper, we named our solution as *RaCE* scheme.

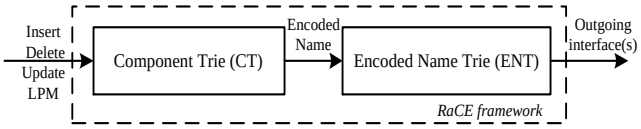

Fig.2. *RaCE* Framework

A. The RaCE Framework and Data Structures

The *RaCE* scheme is using use *RT*, a space-optimized ordered trie to speed up the name lookup and to reduce the total memory consumption. *RT* is path-compressed trie in which nodes with only one child are merged together.

Fig. 2 shows the framework of *RaCE* scheme which comprises of Component Trie (*CT*) and Encoded Name Trie (*ENT*) along with one stack (Available Token Stack (*AT*)) and one Token Frequency Map (*TFM*). *CT*, *TFM* and *AT* together create Component Encoded Module (*CEM*) which encodes each component of a name and maintains their traces. These encoded names are passed to next module Encoded Name Module (ENM) which stores encoded names in the *ENT*.

Fig. 3(a) shows the logical structure of FIB and encoding of the names using the *CEM* module. Fig. 3 (b) shows the FIB structure using *RaCE* scheme. When an *Ipkt* arrives at the router, *CEM* generates <component, token> tuple for each component of that name and generates the encoded name. It is inserted in the *ENT*, if not present. To delete, update and to find LPM, *CEM* acts similar to insert operation.

In next sub-sections, we discuss component encoding and encoded name formation and the lookup operations, such as insert, search, delete and LPM.

Name prefixes	***Encoded Name***	***Outgoing interface(s)***
/in/co/google	*/1/2/3*	
/in/co/google/books	*/1/2/3/4*	
/uk/co/google	*/5/2/3*	
/com/college/academic	*/6/7/8*	
/com/video/movies	*/6/9/10*	
/com/video/movie/lyrics	*/6/9/11/12*	
/com/college/academic/books	*/6/7/8/4*	
/in/co/yahoo/news	*/1/2/13/14*	
/in/college/about	*1/7/15*	
/com/college/CS/news	*/6/7/16/14*	
/com/college/CS/btech	*/6/7/16/17*	
/in/college	*/1/7*	

Fig. 3 (a). FIB table with the name prefixes and their encoded names

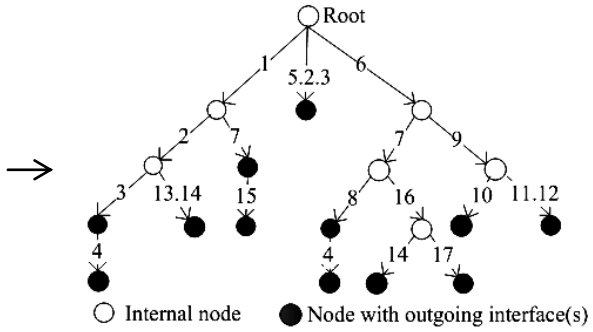

Fig. 3 (b). FIB structure using *RaCE*

B. Component Encoding Module (CEM)

CT data structure is used to produce unique tokens for each *NC*. *CT* is a path compressed trie in which nodes having one child are merged together as shown in Fig. 3(b). Whenever a content request arrives at the NDN router, *CN* is decomposed into its components and a unique integer is assigned to each component. Then, these encoded components form the encoded name and passed to next module *ENM*. The same procedure is repeated for all *CT's* operations - search, insert, and delete of component(s). In *RaCE*, token distribution process maintains the uniqueness among the *NCs*. We define uniqueness in token distribution process in CEM as a property in which CEM ensures that a unique token is assigned to each name's components and identical components share the same token.

C. Encoded Name Module (ENM)

ENT is also space optimized, compact prefix trie which stores encoded names. *CT* passes encoded names to *ENT* which is searched in the *ENT*. *ENT* structure supports four main operations - to search, insert and delete an encoded name, and to find LPM of encoded name. Encoded name is inserted in the *ENT* if already not present in the *ENT*, otherwise encoded name is discarded. To delete an encoded name, *ENT* deletes corresponding components path (not shared by other encoded names) from the leaf node and de-allocates memory. To find the LPM, an encoded name is searched from the root node and matched one-by-one components from longest-to-shortest prefix to find the node with outgoing interface(s).

Radix trie has an important property, called *compaction*, which is followed by CT and ENT. In compaction, a node having only one child, is merged with its parent node. In our proposed solution, we opted this property with a modification that a single child node is merged with its parent node if child node is not containing any index entry. Compaction is performed with every insertion function, if needed.

Fig. 4 and 5 shows the insert and delete operations in ENT through the examples. Suppose a name prefix arrives at the NDN router with the *CN /in/college/about*, propagated by the routing protocols. CT will generate the encoded name /1/7/15 and pass it to the NET. If encoded name is already present in the NET (see Fig. 4), then no operation will be performed. Furthermore, if a request arrives at the NDN router with the *CN /in/college/about/director* which does not exists in the NET, then its encoded name 1/7/15/18 will be inserted in the NET. *RaCE*

repeats the same process for each name prefix required to maintain in the FIB.

Whenever there is need to delete a name prefix from the FIB, CT provides the encoded name to NET. Encoded name is deleted from the NET from leaf node to root if and only if encoded name is not shared by other names of the trie. Suppose, the name to be deleted from the NET is /com/video/movies and encoded name is /6/9/10. The token '10' is not shared by any other encoded name so it can be deleted while other tokens '6' and '9' cannot be deleted as they are shared by other encoded names (see Fig. 5).

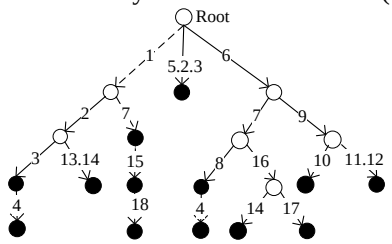

Fig. 4. Inserting an encoded name in the NET

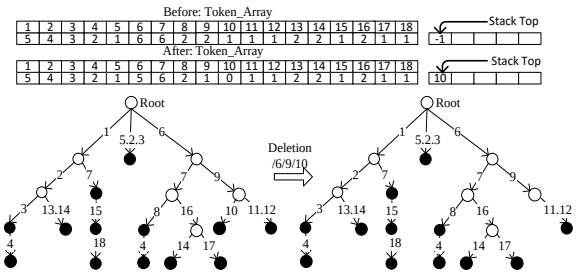

Fig. 5. Deleting an encoded name from the NET

D. RaCE Algorithm

In this section, we present our algorithm for insertion, deletion and LPM using the *RaCE* scheme.

1) Assumptions and variables:

Table I and II list the variables and functions used for the algorithm of *RaCE* scheme.

TABLE I VARIABLES USED IN *RACE* ALGORITHM

Variable	Significance
N	Name prefix to be insert in the FIB or *CN* of an *Ipkt*
L	Total number of components in N
$\boldsymbol{C_i^N}$	i^{th} component of N, s.t. $\sum_{i=1}^{L} C_i^N = N$
$\boldsymbol{C_{list}^N}$	Array containing the components of N
$token(\boldsymbol{C_i^N})$	A unique integer value assigned to C_i^N
$\boldsymbol{N_e}$	Encoded name for N, s.t., $N_e = token(C_i^N)$ for i =1 to L
$\boldsymbol{T_t^C}$	Number of components identified by a particular token present in the *CT* at any time instant t
$\boldsymbol{D_{list}^C}$	Deleted component list
frt_{stack}	Stack to store tokens freed by deleting corresponding components. Tokens are popped for re-use

2) *Algorithm description:*

The algorithm for the *RaCE* scheme is given in Fig. 6. For the implementation purpose, we have used three data structures - Radix trie (RT), stack, and array. We have already discussed about using RT for CT and ENT structures. Stack is used for storing tokens freed by deleting components associated with it and the array is used to store total count of an individual component present in the requested names stored in FIB.

TABLE II FUNCTIONS USED IN *RACE* ALGORITHM

Functions	Description
decompose(N)	Decomposes N and prepares C_{list}^N
insertCT(C_{list}^N)	Searches C_i^N in *CT* and returns token for C_i^N if successful, otherwise inserts C_i^N in CT. Function returns encoded name N_e. to *ENT*
deleteCT(C_{list}^N)	Delete C_i^N from *CT* and re-allocates token to stack
insertENT(N_e)	Searches N_e in *ENT* and returns $FIB_{index}^{N_e}$ for updation if successful, otherwise inserts N_e in *ENT*
deleteENT(N_e, D_{list}^C)	Delete path of D_{list}^C components present in N_e from *ENT*
lpmCT(C_{list}^N)	Searches C_i^N in *CT* and returns token for C_i^N if successful, otherwise stops procedure
lpmENT(N_e)	Finds the LPM of the encoded name N_e
searchCT(C_i^N)	Search the C_i^N in the *CT*

***LPM_RaCE* (*N*)**
Invoke *decompose(N)*
send C_{list}^N to *lpmCT*(C_{list}^N)
lpmCT(C_{list}^N)
 $\forall \boldsymbol{C_i^N} \in \boldsymbol{C_{list}^N}$
 if C_i^N is not in *CT*
  break
 else
  $token(C_i^N) \leftarrow searchCT(C_i^N)$
  $N_e \leftarrow N_e \cup token(C_i^N)$
  lpmENT(N_e)

Insertion_RaCE (N)
Invoke *decompose(N)*
send C_{list}^N to *insertCT*(C_{list}^N)
insertCT(C_{list}^N) //Insertion of a component C_i^N in the *CT*
 $\forall C_i^N \in C_{list}^N$
 if C_i^N is in *CT*
  $T_t^C = T_t^C + 1$
 else
  $T_t^C \leftarrow 1$
  Insert C_i^N in the *CT*
  if *empty*(frt_{stack}) = FALSE // free token to reuse
   token(C_i^N) = *pop*(frt_{stack})
  else
   token(C_i^N) = *token*(C_{i-1}^N) + 1
  $N_e = N_e \cup token(C_i^N)$
  Invoke *insertENT*(N_e)
insertENT(N_e) //Insertion of N_e in the *ENT*
 $\forall token(C_i^N) \in N_e$
 if N_e is not in *ENT*
  Insert N_e in *ENT*

Fig. 6. Pseudocode for *RaCE* lpm and insertion algorithm

To find LPM of incoming *CN*, it is decomposed into components using *decompose (N)* function. Then each component is searched in the *CT*, if any component is not present in the *CT*,

the procedure is terminated. Otherwise, it invokes *searchCT(C_i^N)* function to find the token of component and to form encoded name (N_e). Then, it invokes *lpmENT(N_e)* function to search the N_e in the ENT.

To insert a name prefix into FIB, the function *decompose(N)* decomposes *N* into constituent components and passes these component list (C_{list}^N) to the *insertCT(C_{list}^N)* function. The function searches for the presence of each component in the *CT*, and if found, the component count (T_t^C) is incremented by 1 and the corresponding token is returned. If the component is not present in the *CT*, it is inserted in the *CT* and its component count (T_t^C) is initialized to 1. For assigning the token, the function first checks the frt_{stack} for an available/free token to reuse. If any such free token is available in the frt_{stack}, it is allocated to C_i^{CN} by popping the stack. If frt_{stack} is empty, a new token is assigned to the new component by incrementing the last token generated by 1. At the end of the *insertCT(C_{list}^{CN})* function, *CT* generates encoded name (N_e)and invokes *insertENT(N_e)* to insert N_e in the *ENT*. A function *insertENT(N_e)* searches whether N_e is already existing in the *ENT* or not. If N_e does not exist, then *insertENT(N_e)* inserts N_e in the *ENT* and its corresponding outgoing interface(s).

Deletion_RaCE (N)
Invoke *decompose(N)*
send C_{list}^N to *deleteCT(C_{list}^N)*
deleteCT(C_{list}^N) //Deletion of a component C_i^N from the *CT*
 $\forall C_i^N \in C_{list}^N$
 if C_i^N is in *CT*
  $N_e = N_e \cup token(C_i^N)$
  $T_t^C = T_t^C$ - 1
  if $T_t^C = 0$
   Delete C_i^N from the *CT*
   Add *token(C_i^N)* into D_{list}^C and frt_{stack}
 else
  break
 if D_{list}^C is not empty
  Invoke *deleteENT(N_e, D_{list}^C)*
deleteENT(N_e, D_{list}^C) //Deletion of N_e from the *ENT*
 $\forall D_{list}^C \in N_e$
  Delete *token(C_i^N)* from N_e
  De-allocate memory for re-use

Fig. 7. Pseudocode for *RaCE* deletion algorithm

For deleting a name prefix from FIB, first its components are retrieved using *decompose(N)*. Then the component list (C_{list}^N) is passed to the *deleteCT(C_{list}^N)* function. This function sequentially searches for the presence of each individual components (C_i^N) of *N* using the C_{list}^N. If any component from C_{list}^N is not present in the *CT*, the procedure is terminated. If all the components of a name *N* are available in the *CT*, then the T_t^C for all C_i^N present in C_{list}^N are decremented by 1. If T_t^C of any C_i^N becomes 0 (which signifies that this component is not present in any name stored in the *CT*), then the function deletes that particular C_i^N from the *CT* and the unique token assigned to it earlier is added to the deleted component list *(D_{list}^C)* and is pushed to the freed token stack (frt_{stack}). After the generation of the D_{list}^C (if not empty), the *deleteENT(N_e, D_{list}^C)* function is invoked to delete every freed token from N_e (present in the *ENT*) and de-allocates the memory for re-use.

V. Experiment and Results

A. Experimental Setup

RaCE scheme has been implemented using Java™ programming language which runs on a server with Intel(R) Xenon(R) CPU E5-2695 v2 at 2.40 GHz having 128 GB DDR3 RAM with operating environment 64 bit Windows™ 8.1.

We have used four datasets as input. All are online dataset from *Blacklist* [16], *Shallalist* [17], average workload [4], and heavy workload [4]. *Blacklist* consists of 37,36,394 (~4M), *Shallalist* consists of 45,35,777 (~4.5 M), and *average workload* and *heavy workload* consists of 10.000,000 (10M) domain names (URLs). For a better analysis, we have combined these four datasets to form a dataset of 29M. For the rest paper, we refer *Blacklist* dataset as *B_List*, *Shallalist* dataset as *S_List, average workload* as *10M_AL* and *heavy workload* as *10M_HL*. We have done some pre-processing on the available URLs, to convert them into the traditional named data format. E.g., we have converted URLs of available datasets '*google.com'* into named data format *'/com/google'*. For FIB population, we modified the datasets by reducing the number of components in a name. Table III and IV depicts the detail of the datasets used for FIB population. For the performance evaluation, we have compared *RaCE* with three other schemes, NCE, NPT and CCT as discussed earlier. We run each program 100 times to minimize the impact of hardware dependency and then an average is taken to plot the graphs.

B. Performance Metrics

We have measured the total memory consumption as well as time for name lookup operations in the FIB. The main performance metrics used are as follows:

- **Total memory usage / consumption:** Total amount of memory **(**CT and ENT**)** consumed (in MB) to implement the FIB data structure.
- **Frequency (in million/second):** The number of names (in million) inserted/updated/deleted per second.

In the next sub-section, we shall analyze the impact of token distribution using our proposed *RaCE* scheme.

TABLE III ANALYSIS OF NUMBER OF NAME COMPONENTS FOR DIFFERENT DATASETS

Dataset and Dataset Size	# of Names	Number of Names for the Number of Components							
		<=4	5	6	7	8	9	10	>10
B_List	3,736,394	3,521,370	97,019	50,268	26,398	16,016	10,003	6,113	9,208
S_List	4,535,775	4,465,130	32,257	21,042	10,308	4,211	1,693	6,99	435
10 M_AL	10,000,000	4,893,815	2,608,278	1,171,965	1,070,531	227,534	24,949	2,898	30
10 M_HL	10,000,000	3,807,227	2,601,413	1,690,944	1,181,366	545,237	141,589	28,794	3,430

C. *Token distribution Mechanism*

The main aim of token distribution is to reduce the effect of redundant data by allocating a unique token to identical components. Fig. 8 and 9 shows the number of NCs and the total tokens allocated to them for *B_List* and *S_List* datasets. Different names may share some components at same and/or different levels. Therefore, to generate unique tokens for identical components reduce the generation of total number of tokens. It also depicts the ratio of total tokens distributed to the total number of components. *RaCE* scheme has achieved an average compression ratio of approximately 80% in all the datasets.

TABLE IV STATISTICS OF NAME DATASETS - *B_LIST*, *S_LIST*, *10M_AL*, *10M_HL* AND *29M* DATASETS

Dataset and Dataset Size	# of Names	# of Total Components	Average Component Size in Original (Bytes)	Average Encoded Component Size (Bytes)	Average # of Components per Name	# of Nodes in NCE	# of Nodes in CCT	# of Nodes in NPT	# of Nodes in *RaCE*
B_List	3,736,394	9,482,567	12.83	8.33	2.54	2,340,795	11,730,555	2,340,795	2,001,059
S_List	4,535,775	10,576,826	12.89	8.28	2.33	2,879,339	18,129,256	2,879,339	2,641,457
10 M_AL	10,000,000	47,030,831	27.58	12.97	4.70	2,780,325	17,702,732	2,780,325	1,607,424
10 M _HL	10,000,000	51,213,285	40.49	13.96	5.12	494,088	3,197,223	494,088	285,439
29M	28,272,169	1,21,336,533	28.48	13.24	4.29	8,482,468	50,386,700	8,482,468	6,219,196

TABLE V MEMORY REQUIREMENTS BY ORIGINAL, NCE, NPT, CCT, AND RACE FOR *B_LIST*, *S_LIST*, *10M_AL*, *10M_HL* AND *29M* DATASETS

Dataset and Dataset Size	# of Names	Original size (MB)	NCE Size (MB)	NPT Size (MB)	CCT Size (MB)	*RaCE* Size (MB)	Memory Compression Ratio (%)			
							RaCE vs. Original Size	*RaCE* vs. NCE	*RaCE* vs. NPT	*RaCE* vs. CCT
B_List	3,637,394	54.62	36.11	47.80	74.64	25.59	53.14	29.13	46.46	65.72
S_List	4,535,775	66.55	44.78	64.46	113.75	33.38	49.84	25.46	48.22	70.65
10 M_AL	10,000,000	291.55	31.97	46.89	107.42	21.93	92.48	31.40	53.23	79.58
10 M_HL	10,000,000	418.94	5.68	8.38	19.38	3.91	99.06	31.16	53.34	79.82
29M	28,272,169	848.03	115.25	160.68	311.39	85.20	89.95	26.07	46.97	72.68

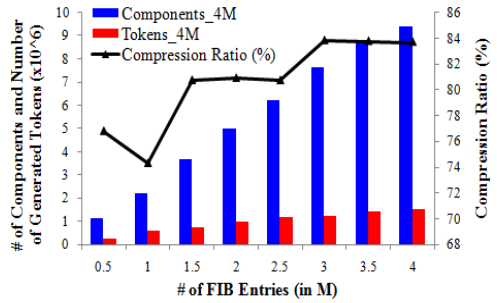

Fig. 8. The number of components vs. number of tokens assigned in the *B_List* dataset

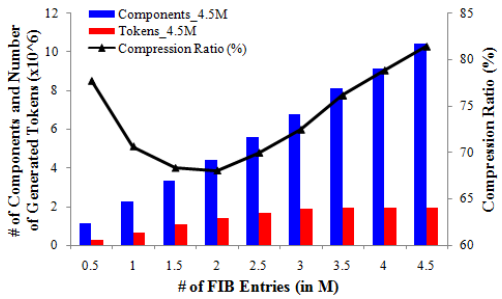

Fig. 9. The number of components vs. number of tokens assigned in the *S_List* dataset

D. *Memory Consumption (worst case)*

We measure the memory consumption for the FIB by using our scheme in compare to the *CCT*, *NPT* and *NCE* schemes and plot the results in Fig. 10, 11, 12, 13 and 14 for *B_List*, *S_List*, *10M_AL*, *10M_HL* and *29M* datasets, respectively. It can be observed that *CCT* and *NPT* consume more memory for implementing FIB than the original size of the datasets. *CCT* and *NPT* data structure consume high memory because each node of the trie maintains additional information as discussed in Section I. On the other hand, *RaCE* scheme consume less memory than other schemes and the original dataset. *RaCE* memory consumption for *B_List*, *S_List*, *10M_AL*, *10M_HL* and *29M* is 38.55 MB, 49.71 MB, 21.93 MB, 3.91 MB and 85.20 MB, respectively. Table V depicts the detail of the memory consumed by different schemes for these datasets. In comparison to *NCE*, *NPT*, and *CCT* schemes, memory consumption in our scheme is reduced by 29.13%, 46.46% and 65.72%, respectively for *B_List* dataset, 25.46%, 48.22% and 70.65%, respectively for *S_List* dataset, 31.40%, 53.23% and 79.58%, respectively for *10M_AL* dataset, 31.16%, 53.34%, and 79.82%, respectively for *10M_HL* dataset and 26.07%, 46.97% and 72.68%, respectively for 29M dataset.

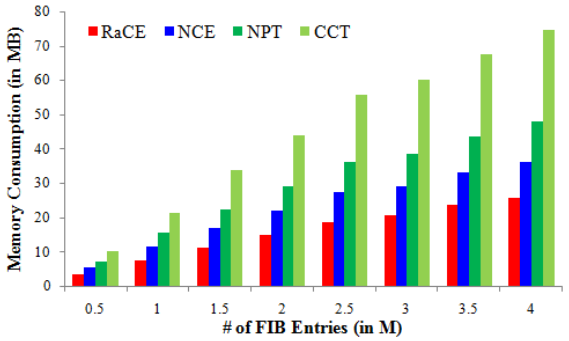

Fig. 10. *B_List* memory consumption - *RaCE*, NCE, NPT, and CCT

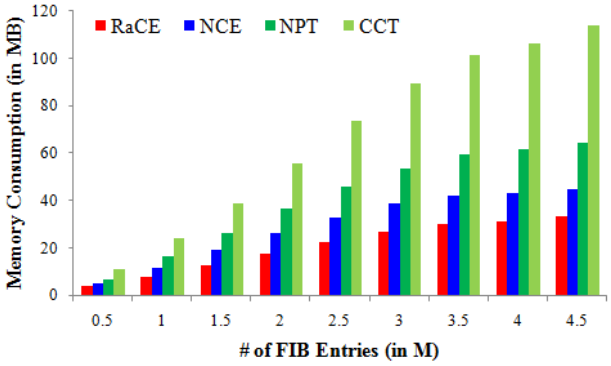

Fig. 11. *S_List* memory consumption -*RaCE*, NCE, NPT, and CCT

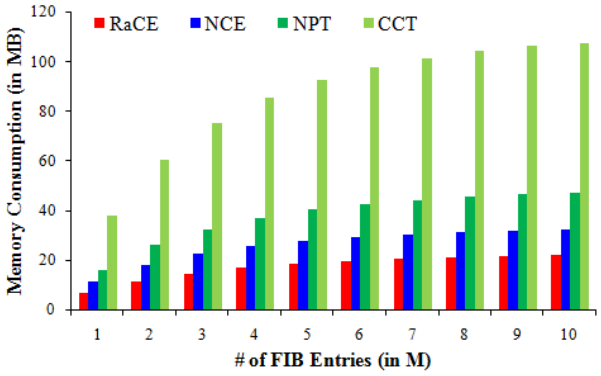

Fig. 12. *10M_AL* memory consumption -*RaCE*, NCE, NPT, and CCT

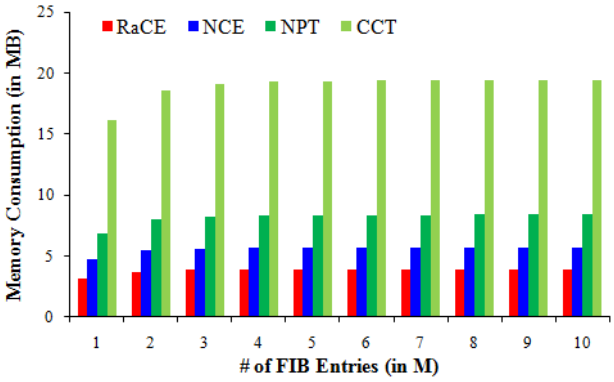

Fig. 13. *10M_HL* memory consumption -*RaCE*, NCE, NPT, and CCT

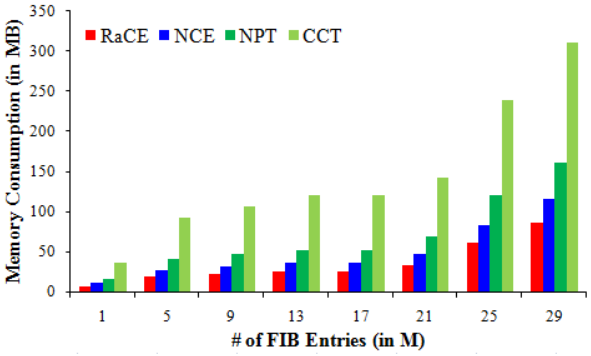

Fig. 14. *29M* -memory consumption - *RaCE*, NCE, NPT, and CCT

E. Insertion, Deletion and LPM Time

In this section, we shall discuss the time required for inserting, deleting and performing LPM. Fig. 15 shows the comparison between the insert, search, and delete frequency (M/s) of *B_List*, *S_List*, *10M_AL*, and *10M_HL* datasets. The total time required to insert, search, and delete all the names using the *RaCE* scheme, for *B_List*, 0.61 M/s, 0.698 M/s, and 0.522 M/s, for *S_List*, 0.638 M/s, 0.72 M/s and 0.55 M/s, for *10M_AL*, 0.657 M/s, 0.708 M/s and 0.505 M/s respectively, and for *10M_HL*, 0.66 M/s, 0.70 M/s and 0.516 M/s, respectively. Fig. 16, 17 and 18 shows insert, search and delete frequency (in M/s) of *RaCE* with other schemes for *29M* dataset.

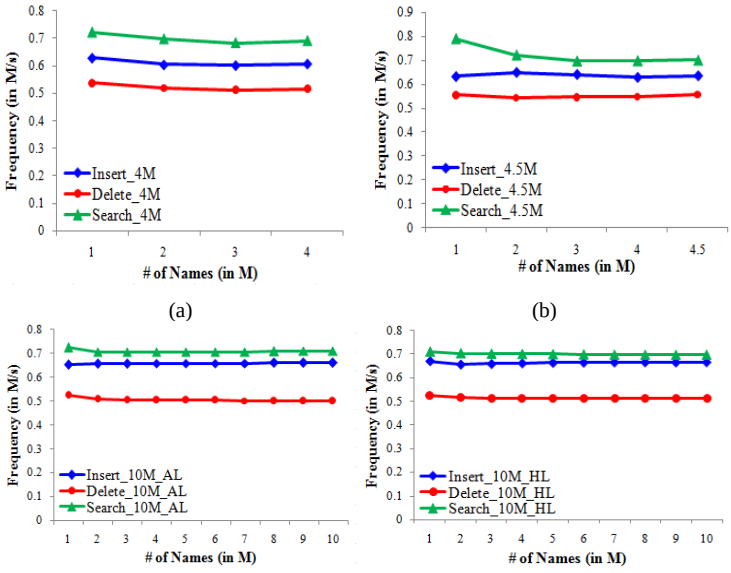

Fig. 15. *RaCE* - Insert, search, and delete frequency (M/s) for the *B_List*, *S_List*, *10M_AL*, and *10M_HL* dataset, respectively

F. Token Re-distribution Performance

In token re-distribution, tokens freed by deleting *NCs* are assigned to new *NCs*. We carry out an experiment to evaluate the efficiency of token re-distribution process. For this, for each 1000 name insertions, we deleted randomly selected single name entry from the *CT* and *NET*. After the component deletion, token can be

re-distributed to the new components. We carried out this experiment for both datasets. Fig. 19 shows the total number of components present in the *B_List* and *S_List* datasets and the distribution of tokens among them.

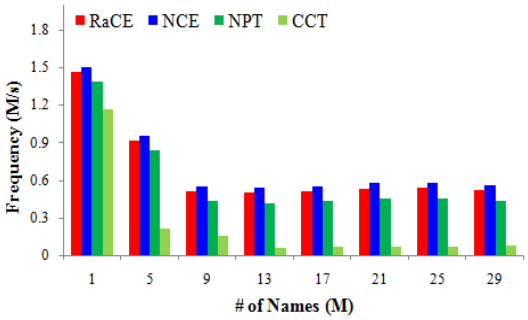

Fig. 16. *29M* - Insert Frequency (M/s) for *RaCE*, NCE, NPT, and CCT

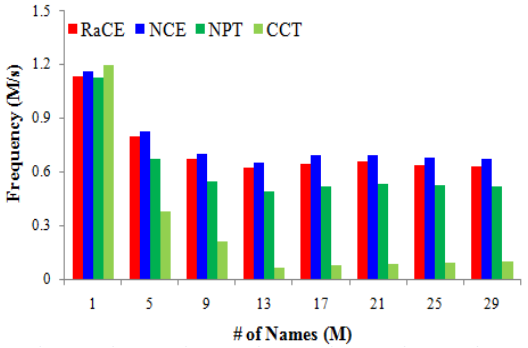

Fig. 17. *29M* - Search Frequency (M/s) for *RaCE*, NCE, NPT, and CCT

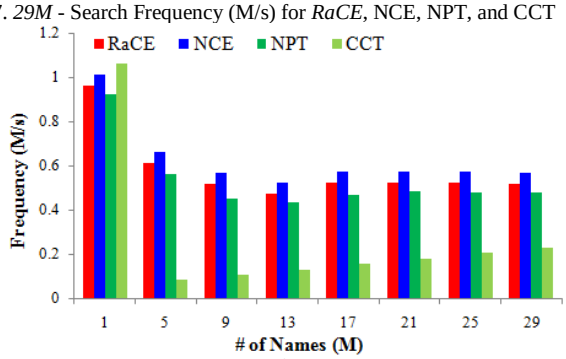

Fig. 18. *29M* - Delete Frequency (M/s) for *RaCE*, NCE, NPT, and CCT

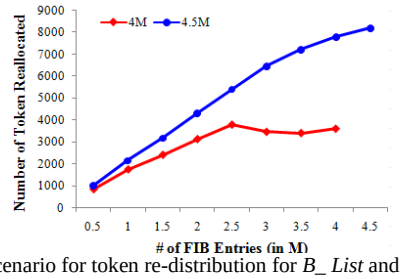

Fig. 19. A real scenario for token re-distribution for *B_ List* and *S_List* dataset

VI. CONCLUSION

This paper discussed a memory-efficient and scalable *NC* encoding based scheme for NDN FIB, called *RaCE*, which is reducing memory consumption of the FIB implementation. The incoming *CNs* or *NPs* are first decomposed in components. Then unique tokens are generated for each component where identical components share same components. The main operations are to insert, delete, and search a name, and to find LPM of name. Extensive experiments show that *RaCE* outperform other existing schemes. In future work, we will try to reduce the insertion, searching and deleting time by introducing the load balancing schemes for the trie.

VII. ACKNOWLEDGEMENT

This work is partially supported by the Alexander von Humboldt Foundation through the post-doctoral research fellow Dr. Vaskar Raychoudhury.

REFERENCES

[1] Saxena D, Raychoudhury V, Suri N, Becker C, Cao J. Named Data Networking: A survey. Computer Science Review. 2016 Feb 21.

[2] Zhang, L. et al. 2010. NDN Project, PARC Technical Report NDN-0001, [Online]. Available: http://www.named-data.net/

[3] Saxena D, Raychoudhury V. 2016. Radient: Scalable, Memory Efficient Name Lookup Algorithm for Named Data Networking. Elsevier Journal of Network and Computer Applications. (Jan. 2016).

[4] Y. Wang, et al., "Scalable Name Lookup in NDN Using Effective Name Component Encoding," In Proc. of IEEE 32nd International Conference on Distributed Computing Systems (ICDCS), 2012, pp. 688-697.

[5] Morrison D. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. J. ACM, vol. 15, no. 4, pp. 514–534, Oct. 1968.

[6] Wang Y, et al., Parallel name lookup for named data networking. In Global Telecommunications Conference (GLOBECOM), (Dec 2011), pp. 1-5.

[7] W. So, et al., "Toward fast NDN software forwarding lookup engine based on hash tables," In Proceedings of the 8th ACM/IEEE symposium on Architectures for networking and communications systems, 2012, pp. 85-86.

[8] W. So, et al., "Named Data Networking on a Router: Forwarding at 20Gbps and Beyond Categories and Subject Descriptors," In Proceeding of ACM SIGCOMM conference'13, 2013, pp. 495–496.

[9] E. Fredkin, "Trie memory," ACM Comm., 3(9), (1960):pp. 490–499.

[10] S. Dharmapurikar, et al., "Longest Prefix Matching using Bloom Filters," In Proceedings of the International Conference on App., tech., architectures, and protocols for computer communications, 2003, pp. 201-212.

[11] Y. Wang, et al., "NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters," International Conference on Computer Communications (INFOCOM), 2013, pp. 95–99.

[12] Y. Wang, et al., "Greedy name lookup for named data networking," In Proceedings of the ACM International Conference on Measurement and modeling of computer systems, 2013, pp. 359-360.

[13] Y. Wang, et al., "Fast name lookup for Named Data Networking," In Proceedings of the IEEE 22nd International Symposium of Quality of Service (IWQoS), 2014, pp. 198-207.

[14] W. Quan, et al., "Scalable Name Lookup with Adaptive Prefix Bloom Filter for Named Data Networking," IEEE communication, 18(1), (2014): 102–105.

[15] H. Yuan, and P. Crowley, "Reliably Scalable Name Prefix Lookup." In Proceedings of the 11th ACM/IEEE Symposium on Architectures for networking and communications systems, 2015, pp. 111-121.

[16] Blacklist. [Online]. Available: http://urlblacklist.com/. [Accessed on Aug. 2014].

[17] Shallalist [Online] Available: http://www.shallalist.de/. [Accessed on Oct. 2014].