Radish — A Simple Routing Table Structure for CIDR

Kazuhiko YAMAMOTO
Nara Institute of Science and Technology
kazu@is.aist-nara.ac.jp

Akira KATO
University of Tokyo
kato@wide.ad.jp

Akira WATANABE
Hitachi Cable Ltd
akira@rop2.hitachi-cable.co.jp

Abstract

The 4.3BSD Reno release introduced the reduced radix tree for routing table to support variable length addresses and best-match based lookups instead of the hash-based scheme of the 4.3BSD Tahoe release. The lookup and maintenance algorithm of Radix is, however, complicated due to support of non-continuous subnet-mask and it is not easy to understand the source code. In order to support classless routing in the WIDE Internet, we designed and implemented Radish, a simpler routing table than Radix. Eliminating support of non-continuous subnet-mask, which is unnecessary in the days of CIDR, Radish provides a simple tree structure and is slightly faster than Radix with some lookup strategies. This paper describes the differences between Radix and Radish and evaluates Radish with several search strategies.

1 Introduction

The Internet has originally provided a class structure for the network layer address, which consisted of a network part and a host part [1]. It was possible to determine which bits are corresponding to the network part by giving only the IP address. The routing was mainly performed by the network part of the destination address.

The class based address structure, however, limited the diversity of network size into three categories; huge, large, and small. In the 1980s, it was considered that the IP address space was large enough and that it was a less valuable resource than memory for routing tables in the routers. In order to prevent routing table growth, typical organizations were recommended to obtain a class B network number even if multiple class C numbers were appropriate. This address assignment policy resulted in the development of a subnet architecture that made it possible to scale down the inadequate network size [2]. It should be noted that non-continuous subnet-masks were permitted so that bridged networks could be migrated into router-connected networks without address reassignment.

As the Internet has grown, three scale problems have become serious in the early 1990s; exhaustion of class B space, explosion of routing table size, and exhaustion of whole IP address space. In order to slow down the assignment of class B numbers, the assignment of multiple class Cs to an organization was started. Since it was hard for new organizations to obtain plentiful address space, efficiency of address utilization became significant. So, many router vendors tend to implement variable length subnet-masks (VLSM) [3]. In this environment, non-continuous subnet-masks are no longer practical. As the assignment of multiple class C, of course, accelerated the growth of routing table, CIDR technology was evolved to aggregate a contiguous address block to a single route [4]. In order to support CIDR and VLSM, routers must provide efficient best-match lookup into a large routing table.

The 4.3BSD Reno release introduced the reduced radix tree for routing table instead of the hash-based scheme of previous releases [5]. The reduced radix tree is general enough to support variable length addresses such as OSI NSAP addresses and is powerful enough to perform best-match lookup even if non-continuous subnet-masks were allowed. The lookup and maintenance algorithms are, however, complicated due to support of non-continuous subnet-masks and it is not easy to understand the source codes. It is necessary to redesign the reduced radix tree structure to make it simple so that everyone can understand easily.

WIDE project, a research oriented network service provider in Japan, has operated its network whose routers are dedicate boxes as well as Sun workstations running SunOS 4.x whose network code is considered to be based on 4.3BSD Tahoe release. So it was essential for the WIDE Internet to support classless routing on SunOS 4.x operating system. For this reason, the authors designed and implemented a simple tree structured routing table that accomplishes best-match lookup. Throughout this paper, we refer to the routing table in the 4.3BSD Reno release as *Radix* due to its filename of source code whereas we call our scheme *Radish* after its simplicity. Both Radix and Radish are variants of TRIE(reTRIEval) [6] and Keith Sklower stated that Radix was a variant of PATRICIA (Practical Algorithm To Retrieve Information Code In Alphanumeric) [7] but such a classification is not essential in this paper. Radish is much simpler than Radix and slightly faster thanks to lookup strategies.

Section 2 reviews how Radix carries out best-match lookup and shows how its operation for tree management is complex in order to support non-continuous subnet-masks. Radish is explained in section 3 from basic concepts

Table 1: Example of routing entries

route	mask
0.0.0.0	0x00000000
133.4.0.0	0xffff0000
133.5.0.0	0xffff0000
133.5.16.0	0xffffff00
133.5.23.0	0xffffff00

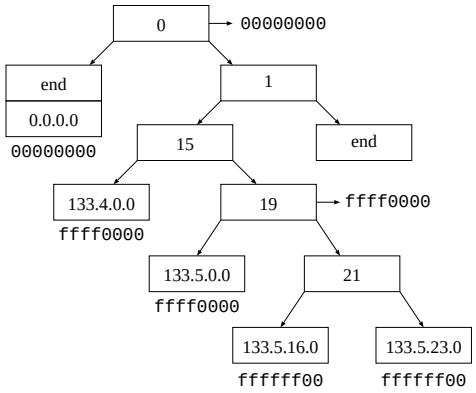

Figure 1: Radix routing table corresponding to Table 1

to those more advanced. This section also contains evaluation of several lookup strategies comparing with Radix. We state current implementation status in section 4 and give a conclusion in the last.

Throughout this paper, we describe a route in (*address*, *mask*) notation as well as address prefix notation in *address*/*masklen*. For example, (133.5.16.0, 0xffffff00) is equivalent to 133.5.16.0/24.

2 Radix

This section first reviews Radix described in [5]. Three examples are given for comprehension of its lookup algorithm. Then we claim to complicated lookup and tree maintenance procedure of Radix.

2.1 Radix tree structure

Best-match lookup, sometimes called longest match, is to find a matched entry in which the number of bit-set in the mask is the maximum. For example, while a destination of 133.5.16.2 matches (0.0.0.0, 0x00000000), (133.0.0.0, 0xff000000), (133.5.0.0, 0xffff0000), and (133.5.16.0, 0xffffff00), a route (133.5.16.0, 0xffffff00) is selected since it is most specific.

Radix performs best-match lookup on a binary radix tree with one way branching removed. A Radix tree is build with nodes and leaves. Each node represents a bit position to test and some of the nodes are associated with masks. A leaf has a route with corresponding masks. Let's consider an example of routing entries in Table 1. A Radix routing table corresponding to these entries is illustrated in Figure 1. Note that the tree structure is independent on the order of entry insertion.

2.2 Radix lookup algorithm

The algorithm of best-match lookup for Radix is a repetition of downward search and backtrack. The first downward search starts from the root of Radix tree and seeks a leaf testing a bit on each node. If the corresponding bit of the given destination key is off, go left, otherwise trace right. When we reach a leaf, the destination key is compared with the route in the leaf. If they are equal, the leaf is the answer. This is, so called, a host match. If not, the destination key is logically ANDed with one of masks in the leaf, then compared with the route. If the masked key is equal to the route, the leaf is the answer. This is a network match or a subnet match. If no matches are found on the leaf, backtrack begins. Backtracking is to trace parent nodes until we find a node that contains one or more masks. If a node with mask(s) is found, the node becomes a new root. The destination key is logically ANDed with one of the masks and downward search start to get a leaf. This downward search is repeated for each mask on the subtree unless the answer is found. If all downward searches fail to discover the matched route, backtrack starts again. In this way, Radix's best-match repeats downward search and backtrack until we get to the answer.

2.3 Examples of Radix lookup

Let's consider three examples for best-match lookup into the routing table in Figure 1. The first example is a subnet match with a destination key of 133.5.16.2. To begin with, lookup starts from the root node. Bit 0 is on, so the right link is chosen as the next path. Bit 1 is off, so the left child node is selected. Bit 15 is on, bit 19 is on, bit 21 is off, we thus arrive at the leaf labeled 133.5.16.0. The destination key is compared to the route but does not match. Then the destination key is logically ANDed with the mask `0xffffff00` and since the result 133.5.16.0 equals the route, this entry is the answer.

The second example of a destination key is 133.5.80.9 which performs network match. Bit 0 is on, bit 1 is off, bit 15 is on, bit 19 is on, bit 21 is off, so we reach the leaf labeled 133.5.16.0. Because both the destination key and logically ANDed key with the mask `0xffffff00` are not equal to the route, backtrack occurs. Go up to the node 21 but it does not have masks, so go up again. Since the node 19 has a mask `0xffff0000`, the destination key is logically ANDed with the mask then 133.5.0.0 is chosen as a new destination key. Considering the node 19 is a new root, downward search starts again. This time bit 19 is off, so we get to the leaf labeled 133.5.0.0. The new destination key is equal to the route, this entry is thus considered as a match.

The last example is a default match where a given destination key is 169.11.16.4. The first downward search reaches the leaf labeled 133.5.16.0 but neither the destination key and masked key is equal to the route. So, backtrack starts to get the node 19 as a new node. Given a new destination key of 169.11.0.0, the downward search leads to the leaf labeled 133.5.0.0. Since the new destination is not equal to the route, backtrack starts again. We go up to the node 15 but it does not have masks. We then go up to the node 0, and find a mask `0x00000000`, so it is chosen as a root for the third downward search. The original destination key is logically ANDed with the mask to get a new destination key 0.0.0.0. On the third downward search, bit 0 is off, so we reach the leaf whose route is 0.0.0.0. Since the new destination key is equal to the route, this entry is the answer.

2.4 Difficulty of Radix

It should be noted that Radix can handle non-continuous subnet-masks since nodes are able to hold them. An interesting question is here; What kind of nodes hold mask(s)? The answer is that a node contains meaningful mask(s) found in its subtree only if the node is the highest possible ancestor for the mask(s). Look at Figure 2 which has two examples (A) and (B). The top rectangle is a destination key and the second one is a mask whereas the last is a logically masked key. Arrows indicate a test bit in a given node. If the test bit is on bit-set of the mask, logical AND operation makes no change for the bit(A). So, this node need not hold this mask since it never changes the direction of downward search. In contrast, if the test bit is on bit-unset of the mask, mask operation may cause a change for the bit test(B). So, this node contains this mask. If all masks are continuous, we can describe this rule simply. That is, a node has mask(s) whose length is equal or shorter than its test bit only if the node is the highest possible ancestor for the mask(s). In Figure 2, the node 19 holds mask `0xffff0000` since its length 16 is shorter than 19. And because nodes whose test bit is 16, 17, or 18 don't exist, the node 19 is the highest possible ancestor.

In addition to the complex search characterized by multiple backtracks, procedures of maintenance for Radix tree are also complicated. Let's consider the insertion of a (route, mask) pair into Radix. 4.4BSD Lite manages a Radix tree for masks as well as that of each protocol families. We first insert the given mask to the mask tree to get the bit position B which makes a branch for the mask in the mask tree. We next add the route to the Radix tree concerted with its protocol family. If the route already exists in a leaf, we sort the mask list in the leaf with the new mask in specific order. Otherwise, a new leaf and a glue node is prepared for the (route, mask) pair. Lastly we search the highest possible ancestor from the leaf according to B. Then mask list is sorted with the new mask in specific order on the found node. In this way, Radix becomes much too complicated in order to support non-continuous masks that have been obsoleted in the classless routing environment.

3 Radish

In order to implement classless routing in the WIDE backbone, the authors designed a new scheme of best-match lookup called Radish. Radish design goals are as follows:

- simple

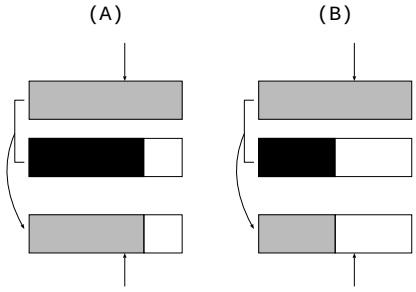

Figure 2: What kind of nodes have masks?

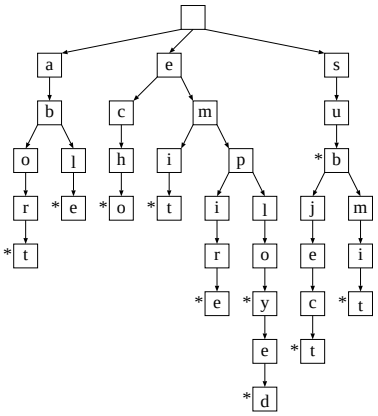

Figure 3: Longest initial substring search

- only support continuous masks
- faster lookup than Radix

We evaluate our scheme focusing on lookup speed since insertion or deletion of entries happens far less frequently than lookups.

Section 3.1 explains a basic algorithm to find the longest initial substring. Section 3.2 and 3.3 describes the tree structure of Radish and its original lookup strategy with some examples. Tree construction of Radish is contained in Section 3.4. Then we evaluate our four lookup strategies in Section 3.5, 3.6, 3.7, 3.8, and 3.9.

3.1 The longest initial substring

Eliminating non-continuous mask, we should redefine the best-match lookup. Here we introduce a new term — "initial substring", which refers to a substring for a given string which starts from the beginning of the string. For example, "employ" is an initial substring of "employee" while "ploy" is not. In the environment where all masks are continuous, the best-match means to find the longest binary initial substring. For example, given a destination of 133.5.16.2, all entries of (0.0.0.0, `0x00000000`), (133.0.0.0, `0xff000000`), (133.5.0.0, `0xffff0000`), and (133.5.16.0, `0xffffff00`) are initial substrings and (133.5.16.0, `0xffffff00`) is the longest one.

Finding the longest alphabetical initial substring is accomplished by an alphabetical radix tree. Figure 3 illustrates an example of alphabetical radix tree for entries {abort able echo emit empire employ employed sub subject submit}. The asterisk mark means that the node represents the end of a word. For example, "sub" is in the dictionary while "subj" is not.

We start searching the longest initial substring from the root with a pointer which points to a recent candidate. The pointer is null at the beginning. While vertexes exist for a given search key, we trace the links downward checking whether or not the current vertex has an entry. If an entry exists, the pointer is modified to indicate the vertex. When the trace breaks, the vertex indicated by the pointer is the answer.

3.2 Radish routing table

The Radish routing table is a binary radix tree to find the longest initial substring. To save memory and to improve lookup performance, the binary tree is reduced. Any vertex which is not associated with a route and does not have two children is removed from the tree. Each vertex has (route, mask) pair. A vertex is marked if a route is associate with it. A mask means which portion of route is valid and indicates a bit to test for branch. Figure 4 is an example of Radish routing table corresponding to routes shown in Table 1. Note that the tree structure is independent on the order of entry insertion.

The best-match lookup is carried out as a variant algorithm of the longest alphabetical initial substring described in section 3.1. We start searching the longest route from the root with an pointer. While vertexes exist for a given

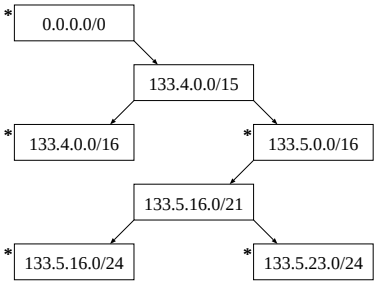

Figure 4: Radish routing table corresponding to Table 1

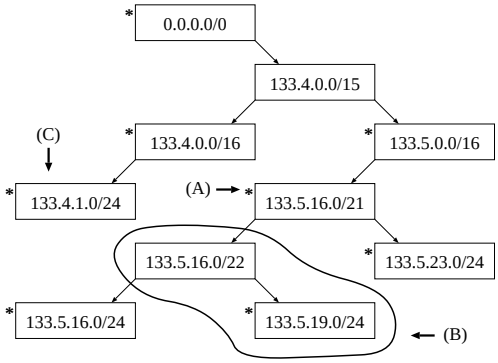

Figure 5: Tree construction of Radish

destination, we trace the link downward. On each vertex, we make two tests. One is to see if we are on a initial substring. Since Radish is a reduced tree, we may skip comparisons for bits in the gap between parent's mask and the vertex's mask. So just one comparison for the bit ($mask - 1$) is not sufficient. The comparison should take into account at least all bits between the parent's mask and the vertex's mask. This can be accomplished by comparison of the whole route and the whole masked destination. If they are same, we are on a initial substring, and if the vertex is marked, we move the pointer here. If they are different, we break the search, and the vertex indicated by the pointer is select as the answer.

After the first test, we make the second test for the bit $mask$ to determine which link we should go down. If the bit is on, go right, otherwise, go left. When the vertex has no child to visit, the answer is the vertex indicated by the pointer.

3.3 Examples of Radish lookup

Let's describe this lookup algorithm using the same examples in section 2. First consider a destination of 133.5.16.2. We start lookup from the root whose route is 0.0.0.0 and mask is `0x00000000`. The destination is logically ANDed with the mask to get masked value of 0.0.0.0. Since the root vertex associates with a route and the masked value equals to the destination, the pointer is updated.

Then test bit 0 of the given destination and go right to a vertex labeled 133.4.0.0/15. For this vertex, the masked value and its route are the same but not marked. So we do not move the pointer and test bit 15. Likewise we trace right, left, left, then finally arrive the vertex labeled 133.5.16.0/24. During this trip, the pointer moves to 133.5.0.0/16 then to 133.5.16.0/24. Because vertex 133.5.16.0/24 has no children, we stop searching and get the answer 133.5.16.0/24.

The second example is 133.5.80.9. We trace right, right, left, arrive at a vertex labeled 133.5.16.0/21. During this trace, the pointer moves to 0.0.0.0/0 then to 133.5.0.0/16. On the vertex of 133.5.16.0/21, the given destination is logically ANDed to get 133.5.80.0. Because the masked value is different from the destination associated with the vertex, we stop the search and conclude that 133.5.0.0/16 indicated by the pointer is the result.

The last example of 169.11.16.4 is very simple. This destination matches the root then go right. Then 169.11.16.4 is logically ANDed with `0xfe000000` to get 169.10.0.0 which does not equal to 133.4.0.0. So the search ends and we get the answer 0.0.0.0/0.

3.4 Tree construction

Insertion of an entry of (address, mask) pair into the Radish tree is straightforward. For a given entry, we trace down the tree checking whether new address matches a vetex and seeing if the length of the new mask is longer than that of the vertex's mask. These two tests decide the position of where the entry is inserted. Roughly speaking, insertions are categorized into three patterns, just marking a pre-exist vertex, inserting it as a leaf vertex, inserting it with a glue vertex. We do not need to manage an extra radix tree nor find the highest possible ancestor.

Table 2: Time required to lookup routing table for Radix and Radish. Unit is micro second.

Table Sequence	JP Measured	JP Random	FULL Measured	FULL Random
Radix	13.06	18.42	12.60	8.454
Radish FS	22.29	6.318	22.24	8.657

Figure 5 illustrates examples of insertion for for three patterns. In the case that a new entry is 133.5.16.0/21, the same vetex exists in Radish tree, so the vertex labeled 133.5.16.0/21 is just marked(A). For a new entry of 133.5.19.0/24, a glue entry labeled 133.5.16.0/22 is prepared and is inserted between 133.5.16.0/21 and 133.5.16.24. Due to the bit 22, 133.5.19.0/24 becomes the right child of 133.5.16.0/22(B). A new entry of 133.4.1.0/24 is just inserted as the left children of the vertex labeled 133.4.0.0/16(C).

3.5 Lookup Strategies and Evaluation

In order to evaluate our lookup scheme, we used DEC HiNote-Ultra IBM-PC compatible machine equipped with Intel 486/DX4-75 CPU running BSD/OS 2.0. The test program is implemented as a user-level process rather than configured within the operating system kernel.

In the evaluation, we configured with two kinds of routing tables and fed two types of destination patterns for each strategy. Routing table “JP” is from a snap shot in July 1995 of a real routing table maintained in a router in the WIDE Internet, where oversea destinations were represented by the default route. It has 2855 entries. On the other hand, Routing table “FULL” is from a snap shot in June 1995 of a real default-free routing table in the Internet. It has 28524 entries.

In order to provide practical evaluation, we recorded 40583 packets on a router in the WIDE Internet, and then removed repeated destination since IP forwarding routing of 4.4BSD Lite caches one entry. This results in a sequence of 32792 destinations including 652 unique destinations. This sequence is labelled as “Measured”. We also tried a randomly generated destination sequence for each of Class A addresses, Class B addresses, and Class C addresses whose length is totally 10000. and this is labelled as “Random”. We measured the average CPU time in micro seconds after 10000 trials and divided it by the length of the sequence. In the following tables, the numbers shown are the average CPU time in micro seconds for one lookup.

JP/Measured is really practical, so many destinations match leaves of the JP routing table. Most destinations of the Random sequence tend to match with the default route in the JP routing table and does not match any vertex on the FULL routing table.

3.6 Forward search

Since the Radish lookup algorithm is very simple and the number of link chases is less than Radix, we expected the Radish lookup to be faster than Radix. Unfortunately, this is not the case. Table 2 is the evaluation of this strategy.

For convenience, we call our original lookup strategy for Radish *forward search*(FS). Table 2 indicates that the Radish forwarding search is about 1.7 times slower than Radix concerned with “Measured” sequence.

The reason that Radish FS is not fast is due to the fact that address comparison is a heavy operation. To support variable length addresses, Radix and Radish compare the masked destination and a route in byte-by-byte basis. Radish FS makes a address comparison on every vertex toward a leaf, while Radix procedure is mainly a chase of links with some address comparison. It is true that Radish FS is fairly fast if address comparison is performed in a single instruction. Table 3 shows the results of forward search whose address comparison is performed by 4-byte word basis for IP addresses.

3.7 Skipping forward search

FS always compares the masked destination and a route at vertexes on the search path. If address comparison is a heavy job, it is not a good idea to compare address at vertexes which does not associate with a real route. We can

Table 3: Time required to Radish forward search by 4-byte word basis

Table Sequence	JP Measured	JP Random	FULL Measured	FULL Random
Radish FS by word	9.827	3.098	10.07	4.207

Table 4: Time required to skipping forward search

Table Sequence	JP Measured	JP Random	FULL Measured	FULL Random
Radish SFS	13.65	5.306	12.89	6.868

skip address comparisons for such vertexes since they can not be candidates. Although this delays exit of search, total CPU time for lookup becomes faster. We call this strategy *skipping forward search*(SFS). The result of SFS with address comparisons in a byte-by-byte basis is shown in Table 4. SFS provides almost the same performance as Radix.

Consider that the maximum depth of the "JP" routing table is 27. Most of the routes are not aggregated and the default route is included. Therefore, the numbers of routes on a search path is about 2, the root (default) and a leaf. SFS thus omits address comparisons 25 times on a path to one of the deepest leaves. With "FULL" routing table, the maximum depth is 23 and there are many aggregated routes. This yields that several real routes have to be compared on a path. Even in this case, performance of SFS is almost the same as one of Radix.

3.8 Skipping backward search

The next interesting strategy is called *skipping backward search*(SBS). In hosts or campus routers, or even on some backbone routers, most lookups match leaves since most traffic is for local communication. An alternative strategy is to find a leaf vertex without any address comparisons, and to then compare address at the leaf. Note that leaves always have a real route. If the masked destination and the route is same, the leave is the match. Otherwise, we trace back parent vertexes and performs address comparisons only on the vertexes which associate with real routes.

Measured CPU time for SBS is shown in Table 5. SBS is about 1.2 times faster than SFS with "Measured" sequence but about 1.7 times slower with "Random" sequence.

3.9 Strategic search

In the case where a destination matches a leaf vertex, SBS is faster than SFS because SBS never make extra address comparisons. However, if the match is located in the upper portion of the tree, SFS is faster than SBS. An alternative strategy called *strategic search*(SS) is to choose SFS or SBS according to the destination.

Recently, IP address allocation is performed in geographical or topological basis rather than in first-come-first-serve basis [8]. The leftmost octet of an IP address roughly suggests the locality of the traffic. We provide an array

Table 5: Time required to skipping backward search

Table Sequence	JP Measured	JP Random	FULL Measured	FULL Random
Radish SBS	10.81	6.516	10.14	7.701

Table 6: Time required to strategic search

Table Sequence	JP Measured	JP Random	FULL Measured	FULL Random
Radish SS(3)	10.47	5.415	10.52	6.940

of 256 integers and maintain so that the *i*-th element represents the cardinality of the routes whose leftmost octets equal to *i*. For example, in the case of Table 1, 0th element is 1, 133th element is 4, and all other elements are 0.

Given a destination, we first check the array with the leftmost octet of the destination. If the corresponding element is smaller than pre-determined threshold, we execute SFS. Otherwise SBS is selected.

For convenience, SS(x) notation is used in this section to represent strategic search with threshold x. SS(0) is identical to SBS whereas SS(2^{24}) is the same as SFS. We measured SS for the four cases sliding the value of threshold and found threshold 3 is the best. Table 6 shows SS(3) has good results on the four cases.

4 Implementation Status

We have already implemented Radish for SunOS 4.x kernel. Since Tahoe-based SunOS 4.x does not support variable length addresses and our main interest is the Internet, Radish for SunOS 4.x (sometimes called Sun Classless) adopts skipping forward search with address comparison by 4-byte word basis. Most workstations running SunOS 4.x on WIDE backbone are utilizing the Radish routing table and they are running stable.

Radish for 4.4BSD Lite, called Radish Lite, currently runs in the user space application that adopts strategic search with address comparison in a byte-by-byte basis. We're merging Radish Lite into the kernel of 4.4BSD Lite.

5 Conclusion

In the days of classless routing, non-continuous subnet-mask bits that make Radix complex are no longer practical. Eliminating the support of non-continuous subnet-mask, the best-match lookup is equal to find the longest initial substring among all routing entries. We designed and implemented a simple tree structured routing table, Radish, which is a variant of TRIE. The maintenance procedure of Radish is straightforward. We proposed a skipping forward search and a skipping backward search which perform as fast as Radix. Our final decision is a strategic search which chooses skipping forward/backward search with threshold 3.

References

[1] J. Postel, "*Internet Protocol*", RFC791, 1981.

[2] J. Mogul and J. Postel, "*Internet Standard Subnetting Procedure*", RFC950, 1985.

[3] P. Almquist and F. Kastenholz, "*Towards Requirements for IP Routers*", RFC1716, 1994.

[4] V. Fuller, T. Li, J. Yu, and K. Varadhan, "*Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*", RFC1519, 1993.

[5] K. Sklower, "*A Tree–Based Packet Routing Table for Berkeley Unix*", USENIX Winter, pp. 93 – 99, 1991.

[6] D. Knuth, "*The Art of Computer Programming*", Vol 3, Addison–Wesley, Reading MA, 1973.

[7] R. Sedgewick, "*Algorithms*", Addison–Wesley, Reading MA, 1988.

[8] E. Gerich, "*Guidelines for Management of IP Address Space*", RFC1466, 1993.