TinyTricia – A Space-Optimized Patricia Trie For Transparent Access to Edge Computing Services

Josef Hammer, Hermann Hellwagner
Institute of Information Technology, Alpen-Adria-Universität Klagenfurt, Austria
{firstname}.{lastname}@aau.at

***Abstract*—Multi-access Edge Computing (MEC) is an essential piece of 5G telecommunication systems to satisfy the challenging low-latency demands of future applications. Our previous publications argue that edge computing should be transparent to clients. We introduced an efficient solution to implement such a transparent approach, leveraging Software-Defined Networking and virtual IP+port addresses for registered edge services. A core component of our architecture is a *Patricia Trie*, which stores all our virtual IP+port addresses. Unfortunately, most implementations of Patricia Tries are not geared toward use cases with millions of keys where a low memory footprint becomes essential. In this paper, we present *TinyTricia*, a space-efficient open-source implementation of a Patricia Trie for keys up to 256 bits. *TinyTricia* can keep track of up to half a billion (2^{29}) keys $\leqslant$ 57 bits and up to a quarter of a billion (2^{28}-1) keys $\leqslant$ 256 bits with tiny memory requirements. In the latter case, each key can have a data value of any type. Keys $\leqslant$ 57 bits require only 8 to 16 bytes per key (i.e., only 0 to 8 bytes overhead per 57-bit key); for keys $\geqslant$ 58 bits, add the key size (in whole bytes) to these values. Thus, for 16.78 million (2^{24}) 57-bit keys, our solution requires between 128 and 256 MiB of memory. Unlike some other space-efficient implementations, *TinyTricia* allows adding and removing keys at runtime.**

***Index Terms*—Patricia Trie, Multi-Access Edge Computing, Space-Optimized, Low Memory Footprint, Software-Defined Networking, IPv4, IPv6**

I. Introduction

Multi-access Edge Computing (MEC) services and convenient access to them are important building blocks of 5G networks and applications. In [1] and [2], we proposed a solution to *transparently* redirect requests to cloud services to the corresponding services running in local edge hosts. This solution, based on Software-Defined Networking (SDN) capabilities, (i) simplifies the development of applications that use edge computing and (ii) allows existing applications to use edge computing without any modifications.

In our *transparent access* approach, the services to be redirected to the edge are first registered with a mobile edge platform provider (fig. 1), identified by their unique combination of domain name/IP address and port number. The network (i.e., an SDN switch) intercepts any request from a client to a registered service and automatically redirects it to the closest available edge node. The edge service processes the request and responds to the client (UE in 5G terms). To achieve transparency toward the clients, our approach uses the packet filtering and rewriting capabilities of OpenFlow [3] (see fig. 2). We refer to [1] for a description of the fundamental solution and discussions about the limitations and scaling options. Furthermore, we refer to [2] for more details (in particular regarding the performance aspects of our approach), a prototype implementation of our SDN controller based on the *Ryu framework* [4], and an evaluation of our prototype on a real edge testbed.

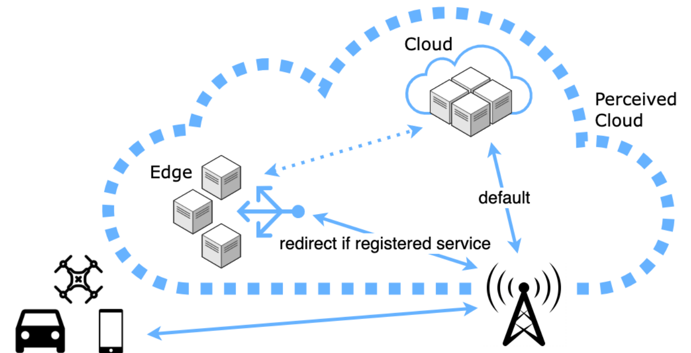


Fig. 1. Perceived cloud vs. real cloud. All requests/responses look like cloud accesses to the client (user equipment; UE) – the redirection to the edge is transparent.

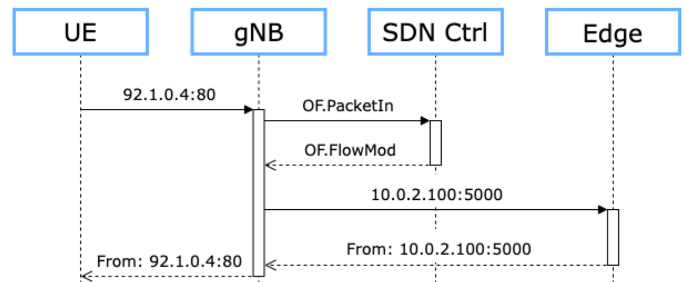


Fig. 2. Routing with a registered service IP: The request is redirected to the closest edge server; transparent to the client (UE). For subsequent requests, the redirection rule is already known to the switch (gNB); the packet is forwarded directly to the edge host (and not to the controller anymore).

One central aspect of our architecture is minimizing the number of flows – for the best utilization of the high-speed and expensive ternary content-addressable memory (TCAM) often used for the flow tables in hardware switches [5]. To be able to detect requests to virtual addresses in our SDN controller, we need to set up separate flows for all destination addresses. However, instead of adding an individual flow for each non-virtual address requested by a client, we use flows with a prefix (i.e., subnet mask) to match as many regular IP

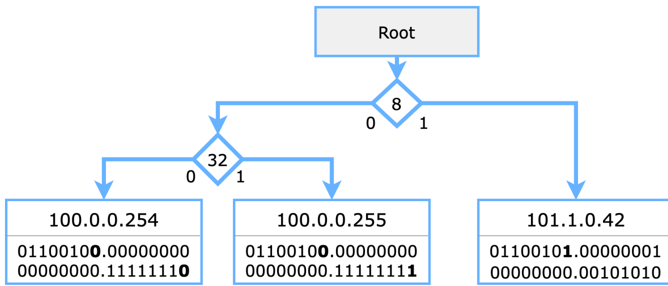
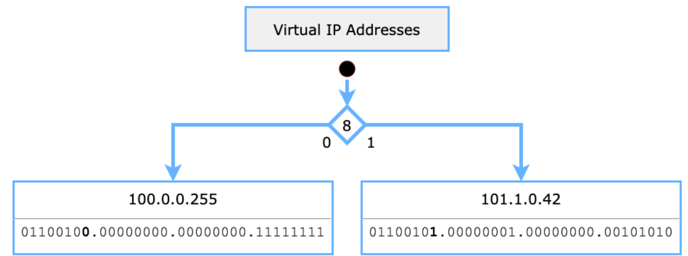


Fig. 3. A simple Patricia Trie containing three 32-bit keys ($k = 32$).



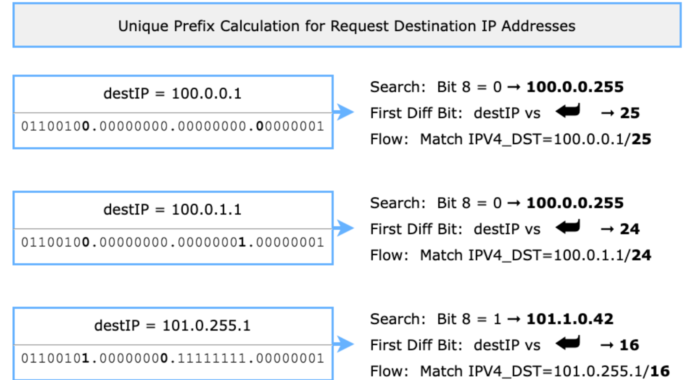


Fig. 4. *Unique Prefix* calculation for a given search address.

addresses as possible. Of course, that prefix must not match any of our registered virtual IP addresses. We call such a prefix the *Unique Prefix* and introduced it in [2]. Using a binary path-compressed search trie (e.g., a Patricia Trie) to store and retrieve virtual addresses lets us quickly determine this *Unique Prefix*. This question would be less straightforward to answer with general (i.e., non-binary) prefix tries ("radix tries"); thus, we decided to focus on Patricia tries for our use case.

A *Patricia trie* (or *Patricia tree*) is a variant of a binary radix trie [6] where the internal nodes only store the position of the first bit ("prefix") that differentiates two sub-tries. As a path-compressed trie, an internal node only exists if necessary, i.e., if at least two keys differ at that specific bit of the key (see fig. 3). Therefore, for a key with k bits, one needs at most k comparisons (a lot fewer on average) to find any element in the trie ($O(k)$). This structure is ideal for storing/retrieving IP addresses where many keys share the same prefix. Thus, routers often use Patricia tries to match an incoming IP address against subnets registered in the trie.

Our use case is slightly different: We use the Patricia trie to efficiently search through (virtual) socket addresses, with the keys being a concatenation of IP address and port number. Therefore, all our keys have the same fixed length: 48 bits ($32 + 16$) for IPv4 or 144 ($128 + 16$) for IPv6. As an additional requirement, our implementation must be able to cope with millions of keys with minimal memory requirements to run it on resource-constrained devices like Raspberry Pis. In particular, we focused on Python-based implementations, as our main project – the SDN controller – is also written in Python. Finally, we needed an implementation that could answer one central question for our work: If a key is not part of the Patricia trie yet, how many of its most significant bits are already included? In other words, with which prefix would our key be inserted? The answer is required to calculate our *Unique Prefix* (see fig. 4 and the section on *Unique Prefix Calculation* in [2] for details). On the other hand, having a data value per key was not a strict requirement for our use case and was thus optional.

Since we could not find a suitable Patricia trie module for our requirements, we developed and open-sourced *TinyTricia* [7], a space-efficient implementation for keys up to 256 bits. With *TinyTricia*, we can keep track of up to half a billion (2^{29}) IPv4 and up to a quarter of a billion (2^{28}-1) IPv6 socket addresses with tiny memory requirements. Unlike some other space-efficient implementations, *TinyTricia* allows adding and removing keys at runtime – which is essential given the online nature of our use case. For 16.78 million (2^{24}) 48-bit keys (IPv4), we would need between only 128 and 256 MiB of memory – ideal for tracking a vast number of keys (e.g., our registered edge services from developers worldwide).

In this paper, first, we survey the related work in section II. We present *TinyTricia's* design in section III and the performance considerations in section IV. Sections V and VI show the limitations and evaluation. Section VII concludes the paper.

II. Related Work

Prefix trees ("Tries") have been around for a long time [8]. Later, Morrison coined the term *Patricia Trie* in [9], where he introduced path compression to store long strings efficiently.

Andersson and Nilsson added level compression in [10], which Nilsson and Karlsson used in [11] for the *LC-trie* – a trie structure that combined both path and level compression for fast IP address lookup. By restricting the node size to 32 bits, their approach, while highly memory-efficient, allows only a small number of keys with a length of at most 128 bits. Kim et al. proposed *FAST* [12], a method of laying out binary search trees in an architecture-sensitive way to improve processor cache line utilization. Similar to *LC-trie*, they also use arrays to avoid the storage overhead required for memory pointers. However, unlike *TinyTricia*, these algorithms require a sorted set of keys as input, and the tries are non-modifiable after construction. That is, both data structures do not support incremental updates and thus are not well-suited

for use in online algorithms. Since Patricia tries have a worst-case search complexity of O(k), one could use a general (i.e., non-binary) prefix trie ("radix trie") to reduce the value of k significantly. However, these tries suffer from poor memory utilization due to partially empty nodes. As an improvement, the *Adaptive Radix Tree* was introduced in [13]. However, their more complex algorithm is less suited for resource-constrained devices. More importantly for our use case, a non-binary prefix trie makes it less straightforward to calculate our *Unique Prefix*.

III. Design

Our main design goal was *low memory consumption* for keys with either 48 (IPv4 + port) or 144 (IPv6 + port) bits – both optionally carrying a data value. This goal led to a design with two slightly different modes: *Regular Mode* and *Compact Mode*. The latter is only for keys up to 57 bits with no data value but provides – as the name suggests – an even better memory utilization. Depending on the specific use case, *TinyTricia* can operate in one of these two modes.

Further goals included a high search performance (see section IV) and versatility: We envisioned *TinyTricia* as a *building block*, not a specialized tool for a specific use case. For example, many existing modules are targeted toward a specific use case like IPv4 or IPv6 addresses – thus not suitable for our keys with 48 or 144 bits. Consequently, *TinyTricia* works with *plain* keys and does not provide specific features like converting textual IP addresses to numbers. These specific features can always be added as an additional layer on top.

This section, however, focuses on our main goal: *low memory consumption*. One of the memory-saving traits shared by both modes is: Nodes do not contain a link to their parent. Thus, during the tree traversal, the algorithm keeps track of the parents for cases where reverse traversal is required (e.g., inserting a new element).

Usually, tree algorithms use nodes with pointers to their child nodes. Pointers make it easy to insert and remove child nodes and do not put a limit on the total number of keys. However, this approach is quite memory-intensive as each pointer requires 32 or 64 bits to be stored (depending on the hardware architecture). Thus, as typical with solutions for small memory requirements, our solution uses an array-based approach that uses indices instead of pointers. Depending on how many total keys shall be supported, these indices require significantly fewer bits than regular pointers.

Fig. 5 shows an example trie (*Actual Patricia Trie*), what it looks like in our implementation (*Logical View*), and how we map the tree to arrays (*Storage View*). We explain the technical details in the remainder of this section.

A. Regular Mode

This default mode is suited for keys up to 256 bits. Since our use case requires keys with up to 144 bits ($k = 144$; IPv6 address + port number), we need at least 8 bits for the prefix value – 0 to 143. Our prefix values only go up to $k-1$ (not k; explained below), leading us to the maximum key length

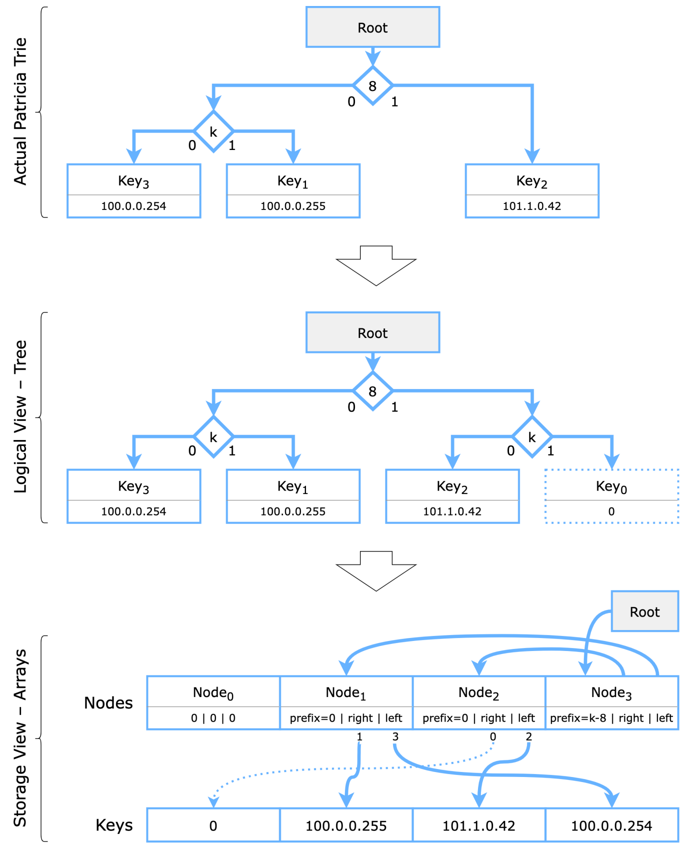


Fig. 5. TinyTricia example with keys added in the following order: 100.0.0.255, 101.1.0.42, 100.0.0.254. For optimal memory utilization, the last (k^{th}) bit is always checked.

of 256 bits. We already decided to omit the usual link to the parent; thus, all our nodes need to store are the prefix and two indices for both left and right child nodes. To best utilize each storage bit, packing all three values into a single integer value proved to be the best option. Therefore, we settled on a 64-bit integer array (called `nodes`): 8 bits for the prefix leave us with 56 bits for both child indices (see fig. 6). With a 28-bit index field, we can address up to 268,435,456 nodes. Since our Patricia trie uses a fixed key length, only leaf nodes may contain keys. Therefore, for n keys, we also need $n-1$ non-leaf nodes – leaving us with 2^{27} possible keys – sufficient for our use cases.

Unfortunately, it is impossible to store a 256-bit key in a 28-bit index field; thus, we need to store the keys in a separate array (`keys`). As with a map, in *Regular Mode*, each key can have a data value of any type, which we store in another array called `data`. Again, the leaf nodes do not store a pointer but the index to the correct entry in the `keys` array – which we call the key's `ID`. In order to save memory, the same `ID` is used for accessing both the `keys` and `data` array.

Storing both keys and data values in arrays separate from `nodes` has an excellent additional benefit: The `nodes` data structure is independent of both the length of the key and the size of one data value. Increasing, e.g., the key length from 64

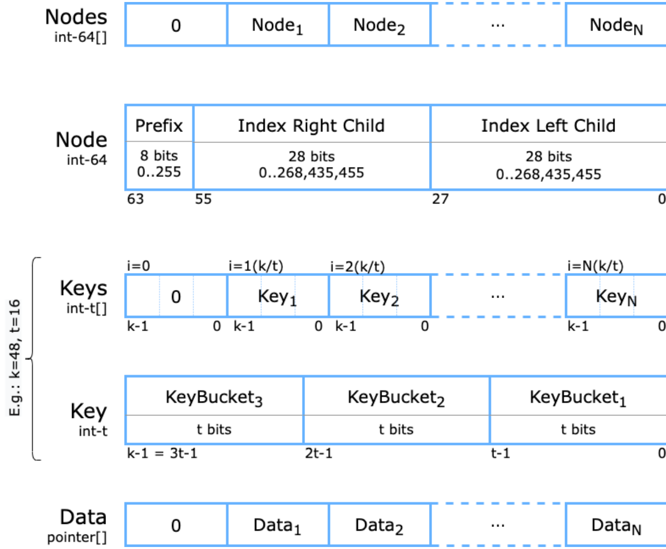


Fig. 6. TinyTricia data structures. `Data` array is optional to save memory.

bits to 256 bits does not affect the `nodes` data structure: each node still fits nicely within one 64-bit machine word. The only thing that needs to be adapted is the size of the items in the `keys` array. Furthermore, the size of one data value is also independent of the length of the key. Moreover, if no values are required, the data value storage can easily be disabled to save memory (using the `keysOnly` parameter).

For performance reasons, our algorithm counts the prefix down to 0 (see section IV on details). Therefore, leaf nodes contain a prefix of 0, and only in this case the child index field contains an `ID` instead of an index to another field in the `nodes` array. However, a leaf node will only point to a single key, leaving one of the two child index fields unused – wasting 28 bits for each key. Consequently, we decided to store the key IDs one level up (basically, at prefix 1) and reduce all prefix values by 1. As a result, for a key with k bits, the maximum prefix value used by our algorithm is only $k-1$ instead of k. Reducing the maximum prefix value by 1 makes a significant difference – it allows for keys up to 256 bits (a multiple of 8 bits) instead of only 255 bits.

Apart from saving memory space, storing two keys in a single leaf node has two other significant consequences. The bad one is that we always need to check the least significant bit of the key, even if there would not exist a branching otherwise. Since our priority is saving memory space, we consider this a reasonable tradeoff. On the other side, the good news is that we can store twice as many keys with the same number of nodes in the best case. *Best case* means that for each key, there is another key where only the rightmost bit differs. In contrast, the *worst case* occurs when for no single key, another key exists where only the rightmost bit differs. The latter would be equal to not using this approach.

For performance reasons, our algorithm uses an empty key_0 (see fig. 5 and section IV). Therefore, the range of possible key IDs is $1..2^{28}-1$. As a result, in the best case, *TinyTricia* can store up to 268,435,455 ($2^{28}-1$) keys in *Regular Mode* – doubling the number of keys compared to storing a single key per leaf node only.

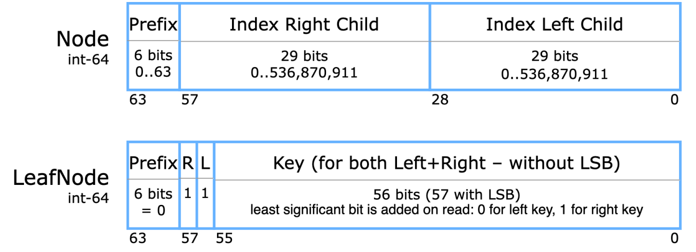


Fig. 7. TinyTricia node structure in Compact Mode (if key length $\leqslant$ 57 bits and no data fields are required). Keys are stored in leaf nodes; the flags `L` and `R` indicate the existence of a left or right key.

B. Compact Mode

For some use cases, one might not need a data value per key but prefer to reduce the memory consumption even more. By disabling the data value storage using the `keysOnly` parameter), one can save another $number_of_keys * sizeof(value)$ bytes. If, in addition to that, the key size is $\leqslant$ 57 bits, *TinyTricia* will automatically switch to *Compact Mode* on initialization. This mode further reduces the required memory and doubles the number of possible keys.

In *Regular Mode*, leaf nodes store IDs in their index fields, which are the indices to the actual values in both the `keys` array as well as the `data` array. However, with both no data values required and a key size $\leqslant$ 57 bits, we can store the keys directly in the leaf nodes instead of any IDs. Consequently, compared to *Regular Mode*, *Compact Mode* reduces the required memory per key by the size of the key (in whole bytes).

While a single 28-bit index field would not provide much space for a key, remember that we always store two adjacent keys (with only the least significant bit being different) in a single leaf node. As a result, we can combine the two index fields into a single key field with 56 bits. Since the two adjacent keys are identical except for the least significant bit, we remove the least significant bit and add it again on reading – 0 for the left key and 1 for the right key. This allows for a maximum of 57 bits per key in *Compact Mode* (see fig. 7). Unfortunately, replacing the two IDs with a key strips us from the ability to use 0 as an indicator for a non-existent key – 0 is a valid key value, and we store two keys in a single field. Therefore, we need two additional bits to indicate whether the key in the field is a valid left or right key. Fortunately, for keys up to 64 bits (as in this mode), a 6-bit prefix is sufficient, which which provides us with the two additional bits required. See listing 1 for how to extract the key from a leaf node.

In the non-leaf nodes, on the other hand, the two gained bits are used to increase the child index fields by one bit each (29 bits). As a result, the number of total keys doubles compared to *Regular Mode*. In *Compact Mode*, *TinyTricia* can store up to 536,870,912 (2^{29}) keys in the best case; half in the worst case.

```
def compactKey(nodeValue, isRight):

    return ((nodeValue & 0xFFFFFFFFFFFFFF) << 1)
            + isRight  # add the lowest bit again
```

Listing 1: Reading the key from a leaf node in *Compact Mode*. `isRight` may be `0` or `1` and corresponds to bit_0 of the key.

Since *Compact Mode* allows keys up to 57 bits, there might be a few unused bits left when used with keys of shorter length. E.g., our use case with 48-bit keys leaves nine bits of the node empty. As a future improvement, one could allow using the remaining bits for data values in use cases where a few bits are sufficient. An example would be boolean flags, e.g., whether a service is already deployed or not. Remember that the remaining bits need to be split between both keys. Thus, in our use case with nine remaining bits, each of the two keys could store up to four flags.

IV. Performance Considerations

When designing TinyTricia, our primary focus was on minimal memory consumption. In addition, we also tried to achieve high search performance by designing our algorithm and data structures in the best possible way. Among our performance-focused design decisions are the following:

Empty first items

If-else branchings are costly since they might trip up the CPU's branch prediction [13]. A simple trick to get rid of such a branching is to start each array with an empty item ($\texttt{node}_0$, $\texttt{key}_0$, and $\texttt{data}_0$). As the most obvious example, in the `search` method, this approach avoids checking for an empty tree. Instead, the root of an empty tree points to $\texttt{node}_0$ – the prefix value of 0 will not enter the loop, and the two child indices with an `ID` of 0 point to the non-existent $\texttt{key}_0$. Since an `ID` of 0 indicates a non-existing key, it follows that both $\texttt{key}_0$ and $\texttt{data}_0$ cannot be used for actual keys and values – thus the empty item in both arrays. Additionally, an empty child index (pointing to $\texttt{node}_0$) automatically indicates a non-existing child node.

Inverse prefix

We decided to invert our prefix, contrary to other algorithms and standard notations. The prefix is usually counted from 1 (left or most significant bit) to the maximum value k (the key's bit length; e.g., 32 for an IPv4 address). However, for our algorithm, it made more sense to count the prefix from $k-1$ down to 0. Consequently, we can compare the stored prefix value against zero, saving an assembly instruction for loading the value of k (see listing 2). Furthermore, we get the current prefix bit of the key with a single bit shift instruction.

V. Limitations

For best optimization, we developed TinyTricia for our specific needs. Therefore, TinyTricia is not a general-purpose Patricia trie, but does have a few limitations.

```
def search(key):

    pos = root
    node = nodes[pos]
    prefix = node >> 56

    while prefix:
        pos = ((node >>
                (((key >> prefix) & 0x1) * 28))
              & 0xFFFFFFF)
        node = nodes[pos]
        prefix = node >> 56

    id = (node >> ((key & 0x1) * 28)) & 0xFFFFFFF

    return id if (id and key == getKey(id)) else 0
```

Listing 2: TinyTricia search method. The prefix is inversed and goes down from $k-1$ to 0. When the tree is empty (Node_0 only) or contains a single node (Node_1 only), the body of the `while`-loop will never be executed.

One significant difference to many other implementations is that we only consider keys of the same fixed length. As a result, only leaf nodes contain keys – there are no partial keys. While the `containsFirstNBits()` method might be a satisfactory solution for some use cases, the design is not targeted toward shorter, partial keys. Furthermore, the algorithm uses integers for the keys; thus, a little-endian machine is required for the algorithm to work correctly. However, one could easily fix this with a conversion layer.

While our data structures allow removing a key at runtime (not implemented yet), reorganizing the arrays can be costly. Therefore, we suggest only removing the pointer to the key but not the key itself. If a use case requires frequent deletion of keys, reorganizing/defragmenting the arrays at specific intervals helps to keep the memory footprint low.

VI. Evaluation

For our research on edge/fog computing we deployed a real testbed named *Carinthian Computing Continuum* (C^3; see fig. 8) presented in [14]. C^3 [15] aggregates a large set of heterogeneous resources and includes an edge computing layer. The entry point to the edge layer is the *Edge Gateway System (EGS)*, a 64-bit x86 system with an AMD Ryzen Threadripper 2920X (12 cores, 3.5 GHz, 32 GiB memory) running Ubuntu. The other edge resources are 35 Raspberry Pi 4B running Raspberry PI OS and five Jetson Nano devices running Linux for Tegra – all with four cores and 4 GiB of memory. Additionally, we also include a few Raspberry Pis 4B with only 2 GiB of memory for specific tests.

We designed our algorithm for fast lookup speed, and performance-critical parts like the search method consist mainly of a few bitwise operations. While we compared the search times with similar algorithms like *py-radix* [16] and *pytricia* [17], presenting any comparisons with these C-based implementations does not make any sense at present, since

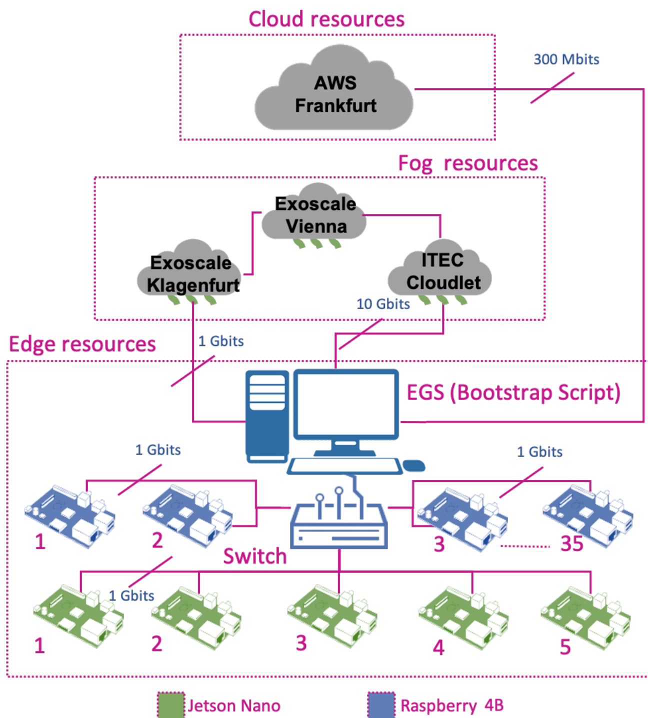


Fig. 8. Carinthian Computing Continuum (C^3).

our algorithm currently exists only as a Python-based implementation. There was one significant difference, however, that we noticed. We tested the implementations up to the maximum number of keys *TinyTricia* can handle: half a billion (2^{29}) keys. On our most resource-constrained devices – the Raspberry Pis with only 2 GiB of memory – both *py-radix* and *pytricia* were killed by the system due to a lack of memory already when tested with only 16.78 million (2^{24}) 32-bit keys. In contrast, our space-optimized *TinyTricia* handled the load well. Remember that for those 16.78 million keys, TinyTricia only requires between 128 and 256 MiB of memory. Thus, running it on a device with only 2 GiB of memory was no issue, while with the two non-space-optimized algorithms, the two 32-bit child pointers alone require that amount of memory.

Note that, in *Compact Mode*, our algorithm is particularly efficient with dense key distributions. This mode stores two keys that differ only in the rightmost bit (bit 0) in a single leaf node. Thus, we recommend modifying the used keys to create a dense distribution where possible. The dense distribution of keys would not only lead to the least memory consumption, but the smaller memory footprint would also increase search performance due to improved processor cache utilization.

VII. Conclusion and Future Work

In this paper we presented *TinyTricia*, our space-efficient implementation of a *Patricia Trie* for keys up to 256 bits. *TinyTricia* is available as open source software [7] and can keep track of up to half a billion (2^{29}) keys up to 57 bits and up to a quarter of a billion (2^{28}-1) keys up to 256 bits with tiny memory requirements. To achieve these tiny memory requirements, our solution uses an array-based approach that allows using indices instead of memory-intensive pointers. While the latter would require 32 or 64 bits per pointer (depending on the hardware architecture), our indices require only 28 or 29 bits (depending on the mode of operation). As a result, *TinyTricia* requires only 8 to 16 bytes per 57-bit key (i.e., only 0 to 8 bytes overhead per key), depending on the distribution of the keys. Thus, for 16.78 million (2^{24}) 57-bit keys, our solution requires between 128 and 256 MiB of memory. Our algorithm is particularly memory-efficient when there are many key pairs with only the least significant bit being different. Unlike some other space-efficient implementations, *TinyTricia* allows adding and removing keys at runtime.

In addition to efficient use of memory, our algorithm is also designed for high speed. Among other things, higher performance was achieved by counting down the prefix to zero (instead of counting up like most algorithms) and reducing the search loop mainly to a few bit shift operations (see listing 2). However, although the algorithm is highly optimized, the Python-based implementation is relatively slow. Therefore, we already designed our algorithm with a C-based implementation in mind, which we plan to provide in the future.

References

[1] J. Hammer, P. Moll, and H. Hellwagner, "Transparent Access to 5G Edge Computing Services," in *2019 IEEE Int. Parallel Distrib. Process. Symp. Work.* IEEE, 2019, pp. 895–898. [Online]. Available: https://ieeexplore.ieee.org/document/8778343/

[2] J. Hammer and H. Hellwagner, "Efficient Transparent Access to 5G Edge Services," in *2022 IEEE 8th Int. Conf. Netw. Softwarization*. IEEE, 2022, pp. 91–96. [Online]. Available: https://ieeexplore.ieee.org/document/9844066

[3] Open Networking Foundation, "OpenFlow Switch Specification v1.5.1," Open Networking Foundation, Tech. Rep., 2015.

[4] "Ryu – Component-Based Software-Defined Networking Framework." [Online]. Available: https://ryu-sdn.org/

[5] A. Shirmarz and A. Ghaffari, "Performance issues and solutions in SDN-based data center: a survey," *J. Supercomput.*, vol. 76, no. 10, 2020.

[6] R. Sedgewick, *Algorithms in C++*, 3rd ed. Addison Wesley, 2002.

[7] "TinyTricia: A space-optimized Patricia Trie (v1.0)." [Online]. Available: https://github.com/josefhammer/tinytricia

[8] E. Fredkin, "Trie Memory," *Commun. ACM*, vol. 3, no. 9, sep 1960.

[9] D. R. Morrison, "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric," *J. ACM*, vol. 15, no. 4, 1968.

[10] A. Andersson and S. Nilsson, "Improved behaviour of tries by adaptive branching," *Inf. Process. Lett.*, vol. 46, no. 6, pp. 295–300, 1993.

[11] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1083–1092, 1999.

[12] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pp. 339–350, 2010.

[13] V. Leis, A. Kemper, and T. Neumann, "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases," *Proc. - Int. Conf. Data Eng.*, pp. 38–49, 2013.

[14] D. Kimovski, R. Matha, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan, "Cloud, Fog, or Edge: Where to Compute?" *IEEE Internet Comput.*, vol. 25, no. 4, pp. 30–36, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9321525/

[15] "The Carinthian Computing Continuum (C³)." [Online]. Available: https://c3.itec.aau.at/

[16] "py-radix: Radix tree implementation (v0.10.0)." [Online]. Available: https://pypi.org/project/py-radix/

[17] "pytricia: An efficient IP address storage and lookup module for Python (v1.0.2)." [Online]. Available: https://pypi.org/project/pytricia/