Fully Distributed Indexing over a Distributed Hash Table

Simon Désaulniers[2(✉)], Adrien Béraud[1], Alexandre Blondin Massé[2], and Nicolas Reynaud[1,2]

[1] Savoir-Faire-Linux, 7275, rue Saint-Urbain, Bureau 200, Montréal, QC H2R 2Y5, Canada
[2] Département d'informatique, Université du Québec à Montréal, Case postale 8888, succursale Centre-Ville, Montréal, QC H3C 3P8, Canada
desaulniers.simon@courrier.uqam.ca

GPG Key Fingerprint: 70B9 F71B 74C9 553D 01A1 A0EF 824A 8B97 F97E 4B08

Abstract. Real-time communication, as well as simple data and file sharing motivates relevant research in network design nowadays. While centralized structures are generally favored over distributed ones for sake of simplicity, a considerable amount of literature has been devoted to the latter. In particular, the problem of distributed indexing is not trivial, and has been addressed by extending classical data structures such as tries, kd-trees. However, all proposed solutions seem to assume that there is a central and unique entity handling the indexing. In this paper, we propose a fully distributed indexing strategy by extending a data structure called prefix hash tree (PHT). More precisely, in this strategy, each node is part of the distributed network and participates in maintaining the distributed index. Our ideas have been implemented in a freely available framework called OpenDHT.

1 Introduction

Since the Internet has become the main foundation of large scale communications, various logical network designs lying on top of this huge network have reached a state of standard. As such, there are networks which rely on a central point of authority providing lower concern complexity when designing client end-user applications. However, those systems lack the ability to scale, they are more vulnerable to straightforward denial of service attacks and have power over the client nodes communication capability. All of these concerns can be avoided or mitigated by using a distributed network design. Indeed, distributed networks have no central authority over a subset of nodes and no central point of failure, and in general, these systems can also easily recover from node failures.

Distributed Hash Tables (DHT) [1,7] are scalable peer-to-peer storage systems providing a simple interface based on the two canonical get/put operations. DHTs are used in a number of applications today where load distribution is an important factor. Those systems are scalable in the sense that they can sustain

E. Sabir et al. (Eds.): UNet 2017, LNCS 10542, pp. 308–318, 2017.
https://doi.org/10.1007/978-3-319-68179-5_27

growing participant load by automatically accommodating for that growth. More precisely, the requests resolution grows logarithmically in the number of nodes [1]. This quality is what makes this type of system of great interest today considering the growing number of participating devices in many networks. Applications include file sharing [2], VoIP [9], large scale website load distribution [12] and many others.

When the only supported operations are `get` and `put`, the implementation of DHTs is straightforward. However, it becomes more involved when other more complex operations have to be supported.

In [4], the authors describe a data structure called *range search tree*, which allows efficient range queries in a fully distributed manner. An alternative data structure called *distributed segment tree* was also considered in [11] to support range queries as well as cover queries. Similarly, partial queries, *i.e.* queries for which the information is incomplete, can be efficiently supported by organizing the data hierarchically [5]. The authors also showed that their strategy is viable in realistic contexts by illustrating it on a bibliographic database.

All kind of queries mentioned in the previous paragraph arise in natural contexts and in most storage access applications, which can potentially include:

- *A distributed media storage* in which access for a subset of media entries based on the name of the author is needed and may be partial (non-exact). Obviously, this use case extends to file searching in general;
- *Distributed scientific computing system* where discovery of resources is based on a parameter being within a certain range;
- *A distributed profile directory* where you obviously search for profiles by first name, last name, city, etc.;

The authors of [8] propose a data structure, called a *prefix hash tree* (PHT), for building an index over a distributed hash table. Their model provides a simple interface for performing "insert" and "lookup" operations on the distributed index. Similarly in [13], the authors use the ideas of the PHT and focus on multidimensional keys aspect by extending the well-known kd-tree data structure. They linearize the keys using traditional projections, following a *space filling curve* like the Z-curve. However, in both cases, very few details are given about the actual implementation and suggest that the indexing data structure is a *centralized* entity sitting on a distributed hash table, *i.e.* that the index itself is not maintained asynchronously by multiple nodes.

Our main contribution consists in proposing an augmented model of the PHT [8] in a fully distributed manner, *i.e* in which the index lies on the DHT and is maintained by the participants performing operations on it. Our construction is inspired from the Kademlia [7] protocol and is freely available as a framework called OpenDHT [10]. We also discuss strategies that were designed to improve significantly the overall efficiency of the basic operations.

2 Motivations

Very few end-user communication applications guarantee complete privacy to their users. In practice, all data about users and their interaction within their personal network is stored in some central database and can be retrieved by anyone with access. In most cases, even the files and messages exchanged between users are not encrypted. Some applications like Signal, a messaging solution, makes a step in addressing these matters, by offering end-to-end and perfect secure encryption of the *content* exchanged. However, it is not possible to hide the *metadata* (who talked with whom, when, how frequently, etc.) [3].

With these concerns in mind, a software called Ring [9] is being under heavy development and aims to offer a completely secured exchange environment. One of the main concerns of the developers is their users' privacy. In particular, while it is virtually impossible to hide all metadata about communications between two users in a given public network, there is a concern about letting the users know exactly what information they share is public or private. More precisely, Ring is a distributed peer-to-peer communication software for chat and VoIP. It uses standard and secure protocols such as TLS (Transport Layer Security) and SRTP (Secure Real-time Transport Protocol). Ring is distributed since all users are part of a DHT network and can establish communication between each other after completing negotiations for NAT traversal using the ICE [6] (Interactive Connectivity Establishment) protocol. This software stands out from the other applications in the sense that no central server is needed and, therefore significantly mitigating privacy concerns associated with threats coming from central authorities.

A diagram illustrating the logical components of Ring is depicted in Fig. 1. The main component we are interested in here is called OpenDHT [10]. This

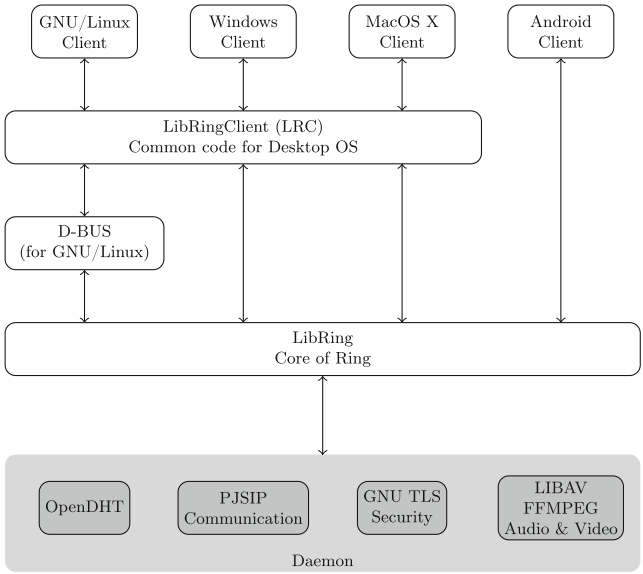


Fig. 1. Architecture of the Ring project.

distributed hash table is a `C++11` implementation of the Kademlia [7] protocol with some novel features such as a `listen(key)` operation for on-going updates about a certain key, a subset of SQL syntax queries capability and a complete cryptographic layer available on its API.

One subproblem encountered during the development of Ring has been the conception of a solution for username directory which is being achieved using a blockchain system called Ethereum [14]. Moreover, a distributed directory for user profiles and a distributed index for DHT nodes proxy discovery is being resolved with the help of the PHT enrichment discussed in the following sections.

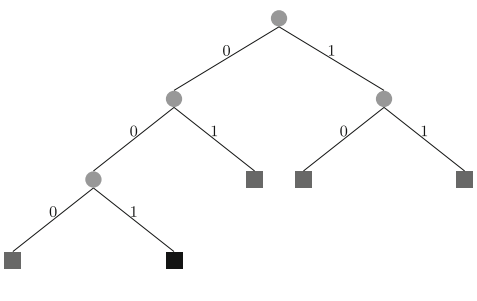
3 Prefix Hash Tree

We now recall from [8] the main ideas around a prefix hash tree data structure (PHT). A PHT is a binary tree, more specifically a *trie*. Every leaf is associated with a unique path down the trie which yields a unique *label*. A label is a prefix of a key $k \in \{0,1\}^D$ for some $D \in \mathbb{N}$. Data is stored only in leaf nodes in such a way that the label of the leaf is a prefix of the key to the stored data. The following constraints must also be satisfied [8]:

1. (*Universal prefix*) Each node has either 0 or 2 children;
2. (*Key storage*) A key K is stored at a leaf node whose label is a prefix of K;
3. (*Split*) Each leaf node stores atmost B keys;
4. (*Merge*) Each internal node contains at least $B+1$ keys in each of its sub-tree.

For sake of simplicity, we assume that the values stored in the data structure are equal to their corresponding key even though in practice, stored elements can be arbitrary complex data.

Example 1. Figure 2 shows potential content of an instance of a PHT. The key `000110` is stored in the leaf labelled `000` and so are all other keys having `000` as a prefix. In particular, the two longest labels of length 3 are `000` and `001` with one of the associated leaves empty (no stored data).



Leaf label	Key
000*	000000
	000001
	000011
	000110
	000111
001*	
01*	010000
	010110
10*	100011
	101011
	101010
11*	111011
	111110

Fig. 2. A PHT storing 12 pairs key-value.

From the picture, we must have $B \geq 5$. Moreover, the labels `000`, `01` and `001` all have `0` as prefix. Therefore, the total amount of keys indexed under prefix `0` can explain the expansion of the left part of the tree. As a consequence, depending on the distribution of the keys, the data structure may become unbalanced, which might have a critical impact on performances.

In its most basic form, the PHT data structure support at least two simple operations, called `lookup(key)` and `insert(key, value)`, which respectively return a view on the data stored in a given node and insert some value with given key. In order not to overload access to higher nodes in the PHT, a binary search scheme is used to find the correct node each time a "lookup" operation is performed. See [8] for the original algorithms and performance analysis.

4 Distributed Indexing

The PHT layer sits on three layers: cryptographic algorithms, the distributed hash table and the transport layer. OpenDHT uses UDP as transport layer. The cryptographic layer will mostly expose asymmetric encryption algorithms which is appropriate to compensate for the untrusted public space in the DHT network.

Indeed, it is worth mentioning OpenDHT is designed for public use. Therefore, significant trade-offs are implied in its design in order to keep a viable and consistent overall behavior. One important design choice that was made consists in preventing anyone to be able to remove values. Instead, a time expiration mechanism was introduced, which implies that all values are stored on the DHT for a limited amount of time. The value lifetime is a key feature used in the PHT layer: When a value has been stored on the DHT, it can only be removed when it expires, according to its time duration specification.

As mentioned before, OpenDHT is an implementation of the Kademlia protocol [7]. Therefore, it uses the XOR metric to perform searches on the key space. In OpenDHT the eight closest nodes to a hash target will be used to perform fundamental operations. At any given time, when some nodes in the eight closest nodes leave, others will take their place as implied by the XOR metric. OpenDHT also features data maintenance to ensure that any content is always available, even when the network topology is significantly modified.

The `listen(key)` operation enables a peer to ask other peers closest to a given hash to yield on-going updates about the associated storage. This operation is used in the PHT layer in order to actively perform elementary rules mentioned in Sect. 3. Figure 3 shows a typical sequence of operations performed according to a "listen" operation. Notice that this operation does not require that the peer A instantiates any further calls after the initial request. This usually last thirty seconds, upon which the *peer A* shall initiate another "listen" request to keep being updated on further changes.

We are now ready to introduce our improvement of the PHT scheme introduced in [8]. Let D be the length of the PHT keys. A *canary* is a value stored in the DHT for the purpose of identifying a node as part of the indexing structure.

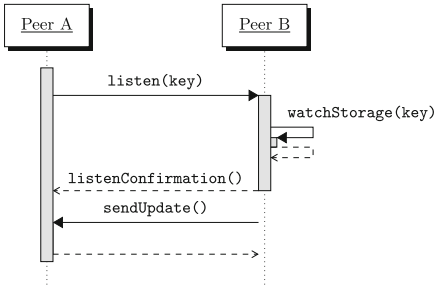


Fig. 3. Listen operation chronological sequence.

Also, let $\ell \in \{0,1\}^m$ be a label with $m \leq D$. If DHT_LOOKUP(ℓ) succeeds and yields a set of values containing at least one canary, then the node labeled ℓ is called a *PHT node*. In particular, it is called a *leaf node* if there is no PHT node either labelled $\ell \cdot 0$ or $\ell \cdot 1$, where $\cdot$ denotes the concatenation, and it is called an *internal node* otherwise.

Based on the definitions introduced in the previous paragraph, algorithms for performing "lookup" and "insert" operations differ from the original algorithms from [8]. Algorithm 1 describes the steps performed for the "lookup" operation. Note that $\text{Pref}_\alpha(k)$ denotes the prefix of length α of a key k. First, two DHT lookups have to be performed to identify the type of the node, which is consistent with how leaf nodes have been defined. Also, it is worth mentioning that the formal definition of a leaf node suggests to perform a third DHT lookup on the other child of the target node. However, this third lookup is not necessary as the algorithm for performing insertion will assure that either both nodes yield a canary or none of them does.

Next, Algorithm 2 describes in details the "insert" operation, where B is the maximal number of values per leaf. Also, it is important to notice that the resulting "leaf" from the call to PHT-LOOKUP is not final. Indeed, by Rule 4, the definition of a leaf node and inherited properties from OpenDHT, merging and splitting PHT nodes is an on-going process which consists in selecting the appropriate node to consider as a leaf based on the number of values yielded by the underlying DHT-LOOKUP operation. We use the canonical "listen" operation from OpenDHT to execute the algorithm again (Line 19) when new PHT node are created below. In addition to inserting the value in the index, the algorithm has to complete the task of maintaining canary values on the network as they are simple DHT values which also expire.

Such strategy for performing concurrently those operations can raise questions about data consistency. The first scenario one can consider when thinking of data inconsistency is data query failure during split and merge operations. The "listen" operation provided by OpenDHT assures high reliability since it is designed to provide quick and light updates. Therefore, when split (or merge) occurs, other indexing participant will quickly react and sync data in new leaves. The time for completing such operation for one index entry is less than a second. Optimizations leading to even lower response time are discussed in Sect. 5.

Algorithm 1. PHT-LOOKUP

Input: A key K
Output: $leaf(K)$

$lo \leftarrow 0$
$hi \leftarrow D$
while $lo \leq hi$ **do**
  $mid \leftarrow (lo + hi)/2$
  $node \leftarrow$ DHT-LOOKUP($\text{Pref}_{mid}(K)$)
  if $mid < D$ **then**
    $node_{child} \leftarrow$ DHT-LOOKUP($\text{Pref}_{mid+1}(K)$)
  end if
  ▷ The type is deduced from the presence of a canary
  $node\_type \leftarrow$ NODE-TYPE($node.values(), node_{child}.values()$)
  if $node\_type =$ "$leaf$" **then**
    return node
  else
    if $node\_type =$ "$internal$" **then**
      $lo \leftarrow mid + 1$
    else
      $hi \leftarrow mid - 1$
    end if
  end if
end while

Algorithm 2. PHT-INSERT

Input: A key K and a value v

$leaf \leftarrow$ PHT-LOOKUP(K) ▷ Might not be a leaf at the end
$\beta \leftarrow$ LENGTH($leaf.prefix()$)
if COUNT($leaf.values()$) $< B$ **then**
  $parent \leftarrow$ DHT-LOOKUP($\text{Pref}_{\beta-1}(K)$)
  $sibling \leftarrow$ DHT-LOOKUP($\text{Pref}_{\beta}(K) \oplus 0 \times 00\ldots1$)
  $count \leftarrow$ COUNT($leaf.values() + parent.values() + sibling.values()$)
  if $count < B$ **then** ▷ Merge
    $\ell \leftarrow \beta - 1$
  else ▷ Straight insert
    $\ell \leftarrow \beta$
  end if
else ▷ Split and insert
  $\ell \leftarrow \beta + 1$
end if
DHT-INSERT($\text{Pref}_{\ell}(K), v$)
UPDATE-CANARY($\text{Pref}_{\ell}(K)$)
DHT-LISTEN($\text{Pref}_{\ell+1}(K)$, PHT-INSERT, K, v) ▷ Listen for new canaries below

Algorithm 3. UPDATE-CANARY

```
1: function UPDATE-CANARY(p: word)
2:     DHT-INSERT(p, canary)                          ▷ Update canary in leaf
3:     DHT-INSERT(p ⊕ 0×00...1, canary)               ▷ Update canary in sibling
4:     x ← RAND(0, 1)
5:     if x ≤ P(p) then                               ▷ Probability of updating the parent
6:         UPDATE-CANARY(SHIFTRIGHT(p, 1))            ▷ Bit shift yields the parent's prefix
7:     end if
8: end function
```

To conclude this section, we describe in details the strategy used to maintain canary values in Algorithm 3. If the data was stored in a centralized database, this step would be straightforward. However, in the distributed case, since different nodes maintain the PHT structure, and since data can expire—in particular canaries—additional care must be taken. We propose a strategy that is very simple to implement and which provides a reasonable guarantee that all canaries are present at all time, at least in the case where the PHT structure is not too unbalanced.

Assume that a call to UPDATE-CANARY() is done for some prefix p. Then we let $P(p)$ be the probability of updating the parent of p. There are two unwanted scenarios that can occur when recursively updating the parent this way:

(i) The root of the PHT, or some internal nodes are not up-to-date;
(ii) The root of the PHT, and some internal nodes, are updated too often.

For instance, if we set $P(p) = 1$, then Scenario (i) never occurs, but the root would be updated as many times as half the number of nodes (a binary tree of n nodes might contain up to $(n + 1)/2$ leaves), which could easily overload it. Obviously, setting $P(p) = 0$ would be worst: Scenario (ii) would never occur, but the PHT would completely disappear. Therefore, it is quite intuitive to set $P(p)$ close to 1. We explore two strategies for handling the updates. For this purpose, let R be the number of times the root is updated.

A first interesting candidate is $P(p) = 1/2$ or a value slightly larger than $1/2$. If the PHT data structure is a perfectly balanced tree, then the probability of not updating the root is

$$P(R = 0) = (1 - P(p)^d)^{2^d} \to 0,$$

where d is the current depth of the PHT. Similarly, the probability of updating the root too often is also small in the perfectly balanced case since $R \sim \text{BINOMIAL}(2^d, P(p)^d)$, so that

$$E[R] = 2^d \times P(p)^d = 2^d \cdot \frac{1}{2^d} = 1.$$

Another interesting strategy consists in setting a probability dependent on the actual depth (or prefix length), such as $P(p) = 1 - 1/2^{|p|+a}$, where $|p|$ is the

length of the prefix p and $a \geq 0$ is some integer parameter. Let K be the set of prefixes that trigger an initial call to UPDATE-CANARY. Then

$$P(\text{not updating the root with prefix } p) = 1 - \prod_{i=1}^{|p|} P(\text{Pref}_i(p))$$

and

$$\begin{aligned} P(R=0) &= \prod_{p \in K} P(\text{not updating the root with prefix } p) \\ &= \prod_{p \in K} \left(1 - \prod_{i=1}^{|p|} P(\text{Pref}_i(p))\right) \\ &\leq 1 - \prod_{i=1}^{D} \left(1 - \frac{1}{2^{i+a}}\right), \end{aligned}$$

the worst case being when there is a single key at depth D. The values of this upper bound can be found in Table 1, with respect to the value of the parameter a and by setting $D = 160$.

Table 1. Upper bound for the probability $P(R = 0)$.

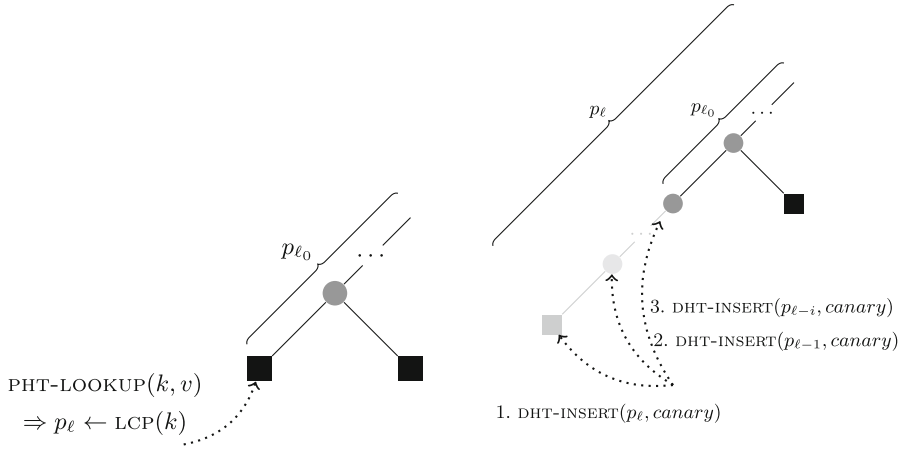
a	0	1	2	3	4	5	6	7
$P(R=0) \leq$	0.7113	0.4225	0.2299	0.1199	0.0613	0.0310	0.0156	0.0078

However, if a is too large, then the nodes close to the root (including the root itself) are updated too often, so that this second scenario can also be problematic. In the future, we intend to study further what would be best to handle the canaries updating and, in particular, what information about the network should be known or estimated in order to control both situations.

5 Optimizations

Our algorithm performing insertion is autonomous in that when a new leaf appears below the node on which a value was first inserted, the old values are automatically "moved" on the new leaf. However, as mentioned by [8], naive implementation of Rule 3 can produce unwanted behavior such as splitting nodes continuously down to the end of the trie. Such scenario can occur if, for instance, B values share a same long prefix of their respective keys. In order to avoid such complications we propose two optimizations. First, double values counting should be avoided when identifying the type of the node based on the number of values yielded by a DHT "lookup". Secondly, splitting should imply the computation of the longest common prefix between all values already present on the

leaf. This prefix indicates the number of PHT nodes to mark with a canary. The participant performing the split shall be the one creating this branch of PHT nodes from bottom to top so that other listening participants shall perform a single PHT-INSERT on the appropriate node. Figure 4 illustrates this process. Note that in this figure, canary update of each siblings of the respective leafs labelled $p_\ell, p_{\ell-1}, \ldots, p_{\ell-i}, \ldots, p_{\ell_0+1}$ is implied. This is needed as per Rule 1.



(a) Longest common prefix (b) Update from bottom to top

Fig. 4. Optimization: split down the trie

In order to provide further enhancements, we propose using a local cache for each instance of indexing participant. Then, subsequent operations in a same region in the tree will benefit from information gathered from previous operations. Algorithms 1 and 2 will use and maintain this cache such that a "lookup" will adjust initial values for *lo* and *hi* variables before commencing.

6 Conclusion

We have proposed a way of maintaining the trie data structure in a distributed manner and according to specific rules inspired by the initial authors of a data structure called PHT. The indexation is assumed to be performed by multiple nodes simultaneously and assures good efficiency and consistency in case that data set is balanced.

Our implementation of our solution is currently still under development. In particular, we are finalizing writing on range query concerns.

In the future, we will provide a more advanced analysis of this strategy in extreme cases such as strongly unbalanced data set. Another angle to address this problem is to remap the data according to a relevant distribution (like Zipfian as suggested by [8]) such that resulting insertions and lookups occur in a mostly

balanced space. Our future work to provide public Ring profiles indexation could require such a strategy. Finally, we plan on providing test and benchmark results in order to support our argumentation on efficiency and data consistency. We hope that our contribution generates more interest and development in the field of distributed systems.

References

1. Balakrishnan, H., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Looking up data in P2P systems. Commun. ACM **46**(2), 43–48 (2003). ISSN 0001–0782. http://doi.acm.org/10.1145/606272.606299
2. Bittorrent: The Bittorrent Protocol. http://www.bittorrent.org/
3. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol (2016)
4. Gao, J., Steenkiste, P.: An adaptive protocol for efficient support of range queries in DHT-based systems. In: 12th IEEE International Conference on Network Protocols (ICNP 2004), 5–8 October 2004, Berlin, Germany, pp. 239–250 (2004). http://doi.ieeecomputersociety.org/10.1109/ICNP.2004.1348114
5. Garcés-Erice, L., Felber, P., Biersack, E.W., Urvoy-Keller, G., Ross, K.W.: Data indexing in peer-to-peer DHT networks. In: ICDCS, pp. 200–208. IEEE Computer Society (2004)
6. Internet Engineering Task Force: Interactive connectivity establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. https://tools.ietf.org/html/rfc5245
7. Maymounkov P., Mazieres, D.: Kademlia: A Peer-to-peer Information System Based on the XOR Metric (2002). http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf
8. Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S.: Prefix hash tree: an indexing data structure over distributed hash tables (2004)
9. Savoir-faire Linux: Ring/ring gives you a full control over your communications and an unmatched level of privacy. https://ring.cx/
10. Savoir-faire Linux: A C++11 distributed hash table implementation (2016). http://opendht.net
11. Shen, G., Zheng, C., Wei, P., Li, S.: Distributed segment tree: a unified architecture to support range query and cover query. Technical report MSR-TR-2007-30, Microsoft Research, March 2007. http://research.microsoft.com/apps/pubs/default.aspx?id=70419
12. Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., Shah, S.: Serving large-scale batch computed data with project Voldemort. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST 2012, p. 18, Berkeley, CA, USA. USENIX Association (2012). http://dl.acm.org/citation.cfm?id=2208461.2208479
13. Tang, Y., Xu, J., Zhou, S., Lee, W.-C.: m-LIGHT: indexing multi-dimensional data over DHTs. In: 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22–26 June 2009, Montreal, Québec, Canada, pp. 191–198. IEEE Computer Society (2009). ISBN 978-0-7695-3659-0. http://dx.doi.org/10.1109/ICDCS.2009.30
14. Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger (2014)