3nuts: A Locality-Aware Peer-to-Peer Network Combining Random Networks, Search Trees, and DHTs

Thomas Janson
Department of Computer Science
University of Freiburg, Germany
janson@informatik.uni-freiburg.de

Peter Mahlmann
Heinz Nixdorf Institute and
Department of Computer Science
University of Paderborn, Germany
mahlmann@uni-paderborn.de

Christian Schindelhauer
Department of Computer Science
University of Freiburg, Germany
schindel@informatik.uni-freiburg.de

Technical Report tr-ri-09-309

December 7, 2009

Abstract

Categorizing peer-to-peer networks from an algorithmic point of view the two extremes of the spectrum are unstructured networks and networks based on plain distributed hash tables (DHT). Unstructured networks stand out with their simplicity, robustness, and support for complex queries. Though, they lack efficient query algorithms providing guarantees. On the other hand, DHT based networks feature efficient lookup algorithms with typically logarithmic hop distance and provide simple and efficient load balancing. Yet, they are limited to exact match queries and in many cases hard to maintain under churn.

In this paper we introduce the 3nuts peer-to-peer network. The aim of 3nuts is to combine unstructured networks and DHTs to overcome their individual shortcomings. This is achieved by cleverly combining self maintaining random networks for robustness, a search tree to allow range queries, and DHTs for load balancing. All network operations in 3nuts are local and distributed, i.e. simple handshake operations maintain the network structure. Efficiency of load balancing, fast data access, and guaranteed robustness are proven by rigorous analysis. Furthermore, 3nuts allows to adopt the structure of the overlay to the physical network with help of local maintenance operations.

1 Introduction

Starting with Napster and Gnutella [7], peer-to-peer networks have become very popular for the exchange of resources (e.g. data) in recent years. In contrast to classical client server architectures nodes (i.e. peers) of a peer-to-peer network have symmetrical functionality: Every peer acts as a server as well as a client. This property bears the potential of excellent failure resilience, since there is no single point of failure and the impact of individual failures may be less than in conventional architectures. On the other hand measurement studies of real world peer-to-peer networks reveal that peer-to-peer networks underly high churn rates [4, 13, 43], i.e. peers frequently join and leave the network without prior notice. Thus, robustness is a key to realize the actual potential of peer-to-peer in applications other than file sharing [41].

By means of their topology peer-to-peer networks may be divided into unstructured and structured networks. A major advantage of unstructured networks, with the prominent example Gnutella [7], is their robustness: Due to the nonexistent structure the topology is easy to maintain under churn. When maintained

with local operations such as introduced in [29] and [30] the evolving networks will be expander graphs w.h.p. and thus are provably robust. On the other hand, a major drawback of the nonexistent structure is the absence of efficient query algorithms. Queries have to be performed by broadcasts of limited depth imposing a lot of traffic or by random walks [28] which reduce network traffic, but massively increase search latency. Structured networks overcome the shortcomings of unstructured networks concerning efficient queries. Starting with CAN [40] numerous network designs based on distributed hash tables (DHT) [23] have been proposed. Prominent examples are Chord [45], Viceroy [31], Pastry [42], Tapestry [18], Koorde [22], D2B [11], and the Distance Halving network [34]. These DHT based networks typically provide logarithmic hop distance[1] for queries and mostly differ in the chosen topology (see Figure 1). So, the design of *static* DHT based networks is well understood [25]. However, DHT based networks are harder to maintain under churn than unstructured peer-to-peer networks in general [6]. Furthermore, the use of hash functions destroys any semantic interdependencies between data: DHTs map data elements to (pseudo) random positions via hash functions so that data items stored at the same peer are usually completely unrelated. Thus, structured queries like range queries can not be performed efficiently using a plain DHT.

A major goal of the 3nuts network introduced here is to combine unstructured networks and DHTs to overcome their shortcomings mentioned above while keeping their individual strength. So, 3nuts uses an extended form of DHTs, i.e. distributed *heterogeneous* hash tables (DHHT) [44], for load balancing and unstructured networks, i.e. directed random networks of constant degree, to achieve robustness under churn. In a nutshell the 3nuts network can be described as follows. In order to preserve semantic relationship of data, peers resemble the prefix search tree (*data tree*) defined by all data available in the network and build the so called *network tree*. In the network tree each node of the data tree is represented by a directed, connected random network of constant degree. Starting with the root of the tree, that is represented by a random graph containing all peers and thus forms a reliable backbone, peers are recursively assigned to subtrees using DHHTs. This recursive assignment is continued until there is only a single peer left in a node or the node represents a leaf of the data tree. Therefore, a peer chooses a path from the root to a leaf of the network tree and participates in each of the random networks representing the corresponding nodes. To allow routing within $\mathcal{O}(\log n)$ hops, each peer maintains so called branch links to a truly random peer of each node of the tree neighboring its own path. 3nuts rigorously forgoes any form of central structures and coordination mechanisms, i.e. the network structure is maintained using local handshake operations applied in the random networks only. Furthermore, one of the major design goals of 3nuts was to forgo the use of heuristics wherever possible. So, the quality of the DHHT scheme used for load balancing has been proven in [44], in [30] it is shown that the random networks used to represent nodes of the data tree are truly random, and in this paper we prove that the lookup operation will need $\mathcal{O}(\log n)$ hops w.h.p. regardless of the structure of the data tree.

The remainder of this paper is organized as follows. In Section 2 we introduce three notions of locality and discuss relevant literature. Section 3 constitutes the main part of this paper and presents the 3nuts architecture in detail. In Section 4 we discuss the realization of locality in 3nuts and in Section 5 we affirm the practical feasibility of 3nuts by experimental evaluation.

2 Locality in Peer-to-Peer Networks

Besides combining concepts of structured and unstructured peer-to-peer networks an aspect of major importance in 3nuts is locality. 3nuts is designed to provide the following three types of locality:

Network locality A peer-to-peer network provides network locality if lookup operations can be performed with small latency. While the typical measure to evaluate routing algorithms is the hop number, this

[1] An exception is the CAN network with hop distance $\mathcal{O}(n^{1/d})$, where d is the dimension of the torus.

Network	Topology	Network locality	Information locality	Interest locality
Gnutella	random graph	no	yes	no
Chord	hypercube	no	no	no
CAN	torus	no	no	no
Viceroy	butterfly	no	no	no
Koorde	de Bruijn	no	no	no
Dist. Halving	de Bruijn	no	no	no
Pastry/Tapestry	mesh of trees	yes	no	no
Skip Graphs	skip list/rings	no	yes	no
Prefix Hash Tree	trie	no	yes	no
DPTree	Skip Graph	no	yes	no
Baton	B^+-tree	no	yes	no
P-Grid	mesh of trees	no	yes	no
SkipNet	skip list/rings	no	yes*	yes*
3nuts	tree/random graphs	yes	yes	yes

Figure 1: Overview of peer-to-peer networks and provided types of locality (*support of interest locality in SkipNet will diminish information locality and vice versa).

alone is not a good measure if the goal is to provide short response times. The reason is that a hop connecting peers in Greece and Australia has higher latency than a hop connecting peers in the same building for example. 3nuts provides network locality by choosing branch links to latency wise close peers.

Information locality A peer-to-peer network provides information locality if closely related data elements are stored on network-wise close peers. In 3nuts this is achieved by using the prefix tree defined by the data as a mold for the network structure.

Interest locality In the Web certain data is intrinsically local, e.g. most of all greek web-sites are created in Greece and accessed from computers in Greece. Hence, it makes sense to store such data on peers located in Greece. 3nuts allows peers to volunteer for the responsibility of particular parts of the data tree and thus implements a basic form of interest locality.

The table shown in Figure 1 gives an overview of peer-to-peer networks and the types of locality they provide. Network designs based on plain DHTs such as Chord [45], Viceroy [31], and Koorde [22], to name some of them, do not support any of the these types of locality. In the following we briefly discuss the relevant literature for each type of locality.

Network locality was the first type of locality that has been addressed by the research community. Pastry [42] and Tapestry [18], based on the seminal work of Plaxton et al. [36], were the first DHT based peer-to-peer architectures providing network locality innately.[2] Other architectures have been extended to support network locality, e.g. [9] and [33] extend the Chord architecture in this regard. The crux in providing network locality is to find latency wise close neighbors without generating too much additional network traffic. In 3nuts this problem is solved by using local maintenance operations applied in the random networks. The scheme used in 3nuts guarantees that the peer with lowest possible latency out of the set of possible neighbors will be found eventually.[3]

[2]We are aware that network locality has also been addressed in CAN [40] and Chord [45]. However, the solutions come with the drawback of loosing the load balancing feature [40] respectively are sketched very briefly [45]. So, we do not consider them here.

[3]Of course, in a dynamic scenario links will not be optimal at any time.

Information locality plays a key role when it comes to support for complex queries such as range queries. The problem of supporting range queries in peer-to-peer overlays has been identified by many researchers, e.g. [16, 19]. Ratnasamy et al. [38, 39] proposed the trie based Prefix Hash Tree (PHT) network, where prefixes of a trie are hashed onto an arbitrary DHT network. This way the load balancing functionality of DHTs can be used. However, DHTs are considered to be inherently ill-suited to range queries [5] and the lookup in PHT is doubly logarithmic in the number of peers.

The skip list [37] based peer-to-peer network Skip Graphs by Aspnes and Shah [3] belongs to the most prominent and first peer-to-peer networks supporting range queries efficiently. Yet, range query support in Skip Graphs is bought dearly by the loss of load balancing: Resources, e.g. data files, are managed by the peer hosting that resource respectively. Consequently, a peer hosting k resources has to maintain k nodes in the Skip Graph overlay. So, in a Skip Graph with n peers and m resources ($m \gg n$ is a typical assumption) a peer has to maintain $\mathcal{O}(k \log m)$ links, whereas the number of links to be maintained in DHT based networks typically is $\mathcal{O}(\log n)$ [45] or constant [34, 31, 22] per peer, and thus independent of the number of resources.

Several tree based peer-to-peer architectures supporting (multi-dimensional) range queries and providing efficient load balancing at the same time have been proposed, e.g. [26, 20, 2, 21, 8, 32, 27, 10]. In the following we briefly discuss P-Grid [2], Baton [20], and DPTree [26]. P-Grid [2] abstracts a binary trie structure defined by the data available in the network. Each peer is responsible for a particular prefix of the trie and maintains links to random peers of every subtree neighboring its own prefix. So far, the structure of P-Grid roughly equals 3nuts' network structure. However, there exist substantial differences between P-Grid and 3nuts. First of all peers sharing a prefix in P-Grid are not connected by random networks. Morever, it is not clear if the links to subtrees selected in P-Grid are truly random, and the load balancing mechanism used in P-Grid is based on heuristics whereas 3nuts makes use of the simple and elegant load balancing provided by DHHTs. Concerning the load balancing in P-Grid it has been critiqued by Ganesan et al. that there is no formal characterization of imbalance ratios and balancing costs [12].

Baton (BAlanced Tree Overlay Network) [20] is based on a binary balanced tree structure. Each peer is responsible for a particular node of the tree. Besides the obligatory parent/child links of a tree, the peers of a layer of the tree are connected by a Chord like ring with pointers to peers in distance 2^i on the ring. Load balancing is achieved by shedding load to an adjacent lightly loaded peer or by a leave/re-join mechanism. In the latter case, the tree may get unbalanced so that it has to be rebalanced.

DPTree (Distributed Peer Tree) [26] is a peer-to-peer architecture inspired by balanced tree indexes (R-Tree [14]). The authors propose to decouple the tree structure from the actual structure of the overlay. This is achieved by using a Skip Graph as overlay structure and choosing peer identifiers such that these represent paths from the root to leaves of the tree structure. Balancing of access load is done with help of a wavelet based mechanism that is used to choose peer identifiers. Peers noticing to be overloaded may shed part of their load to neighboring peers. Unfortunately, a cost analysis of the load balancing mechanism is not given in [26] and it may be critiqued that the authors do not verify the robustness of DPTree under network dynamics. Furthermore, as mentioned by Tran and Nguyen [46], the costs of rebuilding the balanced index tree upon structural changes remain unclear.

Interest locality is rarely addressed in peer-to-peer networks. An exception is the SkipNet [17] architecture introduced by Harvey et al. SkipNet is based on skip lists and therefore closely related to Skip Graphs [3] (SkipNet and Skip Graphs have been developed independently and with different focus). So, it is hardly surprising that SkipNet shares some shortcomings with Skip Graphs, i.e. the lack of load balancing. In fact the authors present a way to provide *constrained* load balancing in SkipNet, but we will not go into detail here and focus on interest locality instead. In SkipNet peers may choose arbitrary name id's. If all peers of a domain (e.g. '.de' or '.it') choose their name id to begin with their domain, then the peers of the same domain will be neighboring in the id space of SkipNet. Using the domain as prefix for data elements also, allows to control data placement. Since the routing algorithm ensures that a query once it reached the target domain will never leave the domain again, SkipNet provides a form of interest locality. This, however,

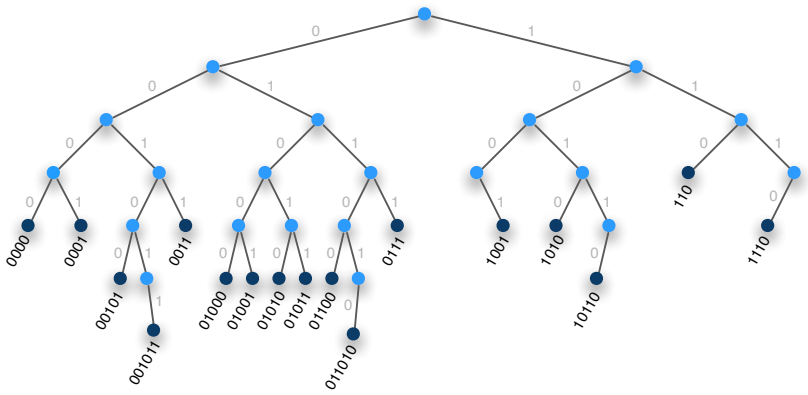


Figure 2: Example of a prefix tree. Nodes storing data elements are depicted dark.

comes at the price of diminishing information locality: To retrieve all documents relevant to a query each domain has to be queried separately then. So, there is a trade-off between information locality and interest locality in SkipNet.

In the bottom line many peer-to-peer architectures supporting either network, information, or interest locality have been proposed in the past years. To the best of our knowledge the 3nuts architecture introduced here is the first peer-to-peer network providing all three types of locality at the same time.

3 The 3nuts Architecture

To cope with the strong dynamics typically arising in peer-to-peer networks and overcome the shortcomings of DHT based overlays when it comes to complex queries, we combine one of the most robust backbone structures with one of the most efficient lookup methods: random graphs and search trees. These two network structures complement one another excellently. On the one hand random networks are provably robust, but there are no efficient lookup algorithms. On the other hand search trees allow efficient and nontrivial lookups like range queries, but are not robust against node failures and do not distribute access load evenly among peers when the tree structure is directly translated into a network, i.e. nodes of the tree are replaced by peers. In this section we describe the overall structure of the 3nuts network, the interplay of tree structure and random networks, a peers local view of the network, and how the network structure is maintained by local periodic handshake operations only.

3.1 Basic Concepts: Data Tree and Network Tree

In contrast to the standard DHT approach where data is assigned to peers, peers are assigned to data in 3nuts. Thus, an existent ordering (e.g. lexicographical) of data is preserved what is essential to realize range queries efficiently. As a consequence, the actual network structure depends on the data currently available in the network and may change when new data is inserted. Before we describe how data forms the tree structure and how peers recreate this tree structure, note that peers in 3nuts only store references to data. The actual data files remain at the peers owning them and these peers inform the peers responsible for maintaining the references for their data regularly.

The *data tree* is the prefix tree (trie) defined by the identifiers of data available in the network (see Figure 2). Principally, any other tree based data structure could be used as well. The main reason why we prefer a trie to more sophisticated balanced trees is its simplicity. Our point is that simplicity can turn out to

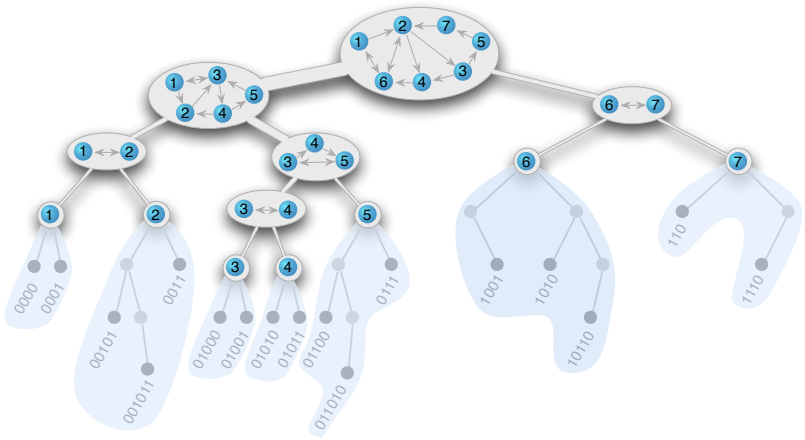


Figure 3: Global view of the network tree.

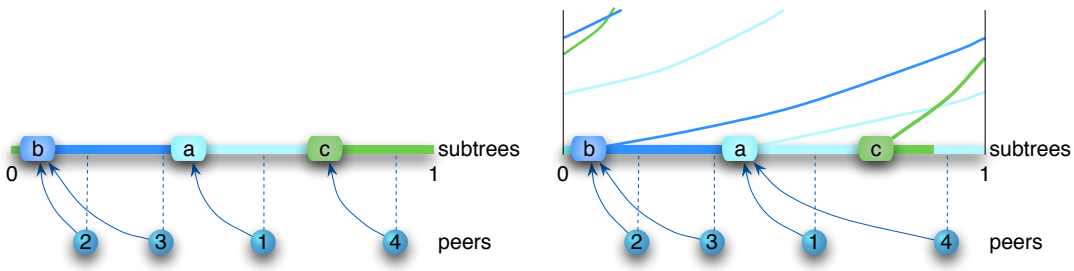
be crucial in a dynamic network under churn and allows to cope with higher churn rates. On the other hand using an unbalanced tree requires more elaborate mechanism to balance the load among peers. Therefore, the load balancing mechanism has to be designed carefully so that it does not dissave the simplicity of the trie. There are many ways to define the load of a subtree, e.g. accesses frequency or number of data elements in a subtree. Here, we use the latter definition of load and balance the number of data elements managed by a peer. Note however, that 3nuts is able to support other definitions of load just as well.

The peers recreate the data tree in a distributed and scalable way. Each node of the tree is replaced by a random network of peers, starting with the root of the tree which is replaced by a random network containing all peers. Then, each of the peers is assigned to one of the child nodes of the root using a simple randomized load balancing mechanism that assigns peers to subtrees with probabilities proportional to the load of the subtree. Peers assigned to the same child node (respectively subtree) then form another random network. This procedure is continued recursively until either a peer is the only one assigned to a subtree or a leaf of the data tree is reached (see Figure 3). For a peer this actually means to choose a path starting at the root of the data tree leading down the tree. We will refer to the combination of random networks and the data tree as *network tree* from now on. The random networks play an important role in 3nuts. Besides making the overlay robust, they are used to spread information about the tree structure, etc., among peers. We will come back to the random networks in Section 3.3 and describe their maintenance in detail.

A peer is responsible for all data in the subtree rooted at the leaf of its path in the network tree (cf. subtrees highlighted light blue in Figure 3). Furthermore, 3nuts allows to store data in internal nodes of the tree. Therefore, each internal node of the network tree has a particular peer that is responsible for managing data in this node. In the following sections we describe in detail the assignment of peers to subtrees, the assignment of responsibilities for internal nodes, the maintenance of random networks among all peers participating in a node of the network tree, and then describe a peers local view of the network tree and see how information about the tree structure is exchanged among peers.

3.2 Peer Assignment, Load-balancing, and Responsibilities

The recursive assignment of peers to subtrees is done using distributed heterogeneous hash tables (DHHT) [44], which is an extended form of consistent hashing, a.k.a. distributed hash tables (DHT) [23], to support non-uniform load distributions. In our case the actual number of peers assigned to a subtree depends on the load, i.e. the amount respectively popularity of data stored in this subtree. We will give a brief description



(a) DHT (b) DHHT

Figure 4: Assigning 4 peers to subtrees a, b, and c using a DHT respectively DHHT (dashed lines show the positions peers have been hashed to in the $[0, 1)$ interval).

of consistent hashing and then describe the weighting extension, in both cases with focus on our particular application. To avoid confusion please recall that we assign peers to data (respectively subtrees) while the usual approach is just the opposite way around, i.e. data is assigned to peers. This allows us to preserve a given ordering of data elements and thus overcome the crucial limitation of DHT based peer-to-peer overlays to exact match queries.

We exemplify the peer assignment in an arbitrary node v of the tree with child nodes $v_1, \ldots, v_k$. Furthermore, we assume that peers $p_1, \ldots, p_n$ have been assigned to v. Consistent hashing uses a "two-sided" hashing into a continuous range $M = [0, 1)$ to assign peers to the subtrees rooted at $v_1, \ldots, v_k$. Peers and subtrees are mapped randomly into M by hash functions h_1 respectively h_2. Then, peers are assigned to the subtree which is closest to them in descending direction in M. So far all peers and subtrees are handled as if they are uniform. While this is reasonable and intended in the case of peers, it is very likely that some subtrees hold more data than others and thus generate a higher load to the peers in these subtrees. So, assigning peers uniformly to subtrees it is highly likely that the load is not spread evenly among the peers.

To take different weights of subtrees into account and make the number of peers assigned to a subtree reflect its weight, the scheme is extended as follows. Let $w_1, \ldots, w_m \in \mathbb{R}^+$ denote the weights of the subtrees rooted at $v_1, \ldots, v_m$ and let $p'_i = h_1(p_i)$ and $v'_j = h_2(v_j)$ denote the position of peer p_i respectively the subtree rooted at v_j in M. Then, we define a scaled distance function

$$L_w(p_i, v_j) = \frac{-\ln\left(\left(1 - \left(p'_i - v'_j\right)\right) \mod 1\right)}{w_j},$$

with $x \mod 1 := x - \lfloor x \rfloor$. Now each peer p_i is assigned to the subtree rooted at the node v_j minimizing the term $L_{w_j}(p_i, v_j)$ (see Figure 4). For a peer p_i and a node v_j we will also refer to the value of this function as *height*.

If we further extend the scheme described so far to use double hashing, then a peer p_i is mapped into M using $p'_i = h_1(p_i)$ as before, but each subtree rooted at v_j, $1 \leq j \leq k$, has an individual hash function h_{v_j}. A node then calculates his heights for each v_j at position $h_{v_j}(p'_i)$ and is assigned to the subtree minimizing the height. Using the combination of DHHT and double hashing the following theorem does hold.

Theorem 1 *Assigning peers to subtrees using the DHHT scheme in combination with double hashing it holds w.h.p. that*

$$\Pr[p_i \textit{ is assigned to } v_j] = \frac{w_j}{\sum_k w_k}.$$

Proof: The theorem is a direct consequence of Theorem 10 in [44]. □

Hence, peers are assigned to subtrees with probabilities proportional to the weights of the subtrees. Note that the runtime of the assignment using double hashing is linear in k, i.e. the number of subtrees. However, in our scenario k is a small constant.

As mentioned in the previous section every node of the network tree has a designated *responsible peer*. The responsible peer has to manage references to data stored in this node, create new subtrees when data is inserted, and delete empty subtrees. The responsible peer for a node v of the tree is the peer that has been assigned to v with the lowest height, i.e. the decision about responsibility is made in the parent node of v. This choice is reasonable since the selected peer will be the last peer to leave the subtree rooted at v if this subtree's load decreases or the load of other subtrees rooted at v's siblings increases. Furthermore, the selection mechanism ensures that the selected responsible peers are truly random and thus the responsibility for internal nodes of the network tree is spread evenly among all peers.

Hashing peers onto subtrees it is possible that no peer is assigned to a specific subtree and thus no peer is responsible to manage the data in that subtree. This will principally happen to subtrees with low weight or leaves of the network tree. If there is such a vacant subtree, then a particular peer is selected to manage this subtree by a mechanism which we call *shanghaiing* (inspired by the english slang term, describing the common act of forcibly conscripting someone to serve a term working on a ship, usually after having been rendered senseless by alcohol or drugs, during the 19th century [35]). Therefor, the peer that has the lowest height for the vacant subtree is selected to be shanghaied. In the scenario of Figure 4(b) peer 4 would get shanghaied to be responsible for subtree c for example. Again, this choice is natural since the chosen peer will be the first to be assigned regularly to the vacant subtree if the weight of the subtree increases. A shanghaied peer is responsible for a subtree until either another peer is assigned regularly to that subtree or a peer with lower height is shanghaied to be responsible for that subtree.

3.3 Maintaining Random Networks

Random graphs play an important role in 3nuts. Their simplicity and provable robustness make them an ideal tool to improve the churn and fault resilience of a network. As we have seen in Section 3.1 all peers that have been assigned to a particular node of the network tree are connected by a random network. Here, we use d-out-regular multi-digraphs and maintain these using the *Pointer-Push&Pull* operation we introduced in [30]. The Pointer-Push&Pull operation is a simple graph transformation that is initiated randomly by each peer periodically and guarantees the resulting graph structure to be truly random.

A single Pointer-Push&Pull operation works the following way: A peer p_1 initiating the operation selects a random neighboring peer p_2. Peer p_2 also selects a random neighbor p_3, replaces p_3 by p_1 in its table of random neighbors and sends the id of p_3 back to p_1. Then, p_1 replaces p_2 by p_3 in its table of random neighbors. So, a Pointer-Push&Pull operation involves only two messages between two peers. In [30] we show that this operation despite of its simplicity guarantees connectivity and, more importantly, generates truly random digraphs as stated in the following theorem.

Theorem 2 *Let G_0^* be an d-out-regular weakly connected edge labeled multi-digraph with n nodes. Then, applying random Pointer-Push&Pull operations repeatedly to the graph will construct every d-out-regular weakly connected edge labeled multi-digraph with the same probability in the limit, i.e.*

$$\lim_{t\to\infty} P\left[G_0^* \xrightarrow{t} G^*\right] = \frac{1}{|\mathcal{MDG}_{n,d}^*|},$$

where t is the number of operations and $\mathcal{MDG}_{n,d}^*$ denotes the set of all d-out-regular weakly connected edge labeled multi-digraphs.

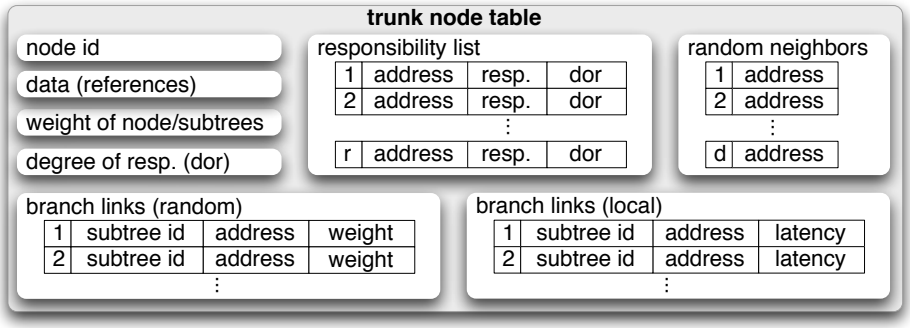


Figure 5: Information a peer holds for a trunk node.

Proof: The theorem is proven in [30]. □

Note that Pointer-Push&Pull operations can be used to replace the mandatory heartbeat (ping) messages used to verify the availability of neighbors in dynamic networks. Therefore, Pointer-Push&Pull operations do not introduce additional traffic to the network.

An important consequence of Theorem 2 is that a peer will see every other peer participating in the same random graph, i.e. node of the network tree, over time. Hence, Pointer-Push&Pull operations constitute an excellent tool to spread information about the tree structure, weights, etc. among peers without inducing additional traffic to the network. This fact is the main reason for us to prefer multi-digraphs over the more common domain of regular undirected graphs, which could be maintained using the related 1-Flipper operation [29].

3.4 A Peer's Local View

In the previous sections we have seen how peers recreate the data tree by choosing their path in the network tree and replacing nodes of the data tree by random networks. A peer therefore only has a limited local view of the network tree. For a peer we refer to the nodes it has been assigned to (either regularly or shanghaied) as *trunk nodes* of the peer. For each trunk node a peer maintains a *trunk node table* (see Figure 5). In this table node id, weight of the node and subtrees, and (if the peer itself is responsible for the node) references to data are stored. Furthermore, the trunk node table contains the following lists:

Responsibility list A list of the r peers with highest responsibility for this node. Responsibility is determined by the height with which a peer has been assigned to this node by the assignment process in the parent node (see Section 3.2). Here, lower height connotes higher responsibility. While it would be sufficient to store one peer responsible for the node, having a list of the r peers with highest responsibility greatly helps to improves the stability of the network under churn.

Random neighbors A list of the d neighboring peers in the random network corresponding to this node of the network tree. This list is used to perform the regular Pointer-Push&Pull operations and thus the list is constantly refreshed and its randomness is guaranteed by Theorem 2.

Branch links To allow efficient routing a peer maintains *branch links* to all subtrees neighboring its own trunk nodes. Here, we distinguish two types of these links: *random* branch links and *local* branch links. The former links point to truly random peers of each subtree. With each of these links the id and weight of the subtree as well as the address of the corresponding peer is stored. Moreover, each link is tagged with an age and a link is replaced whenever a peer participating in the subtree is met

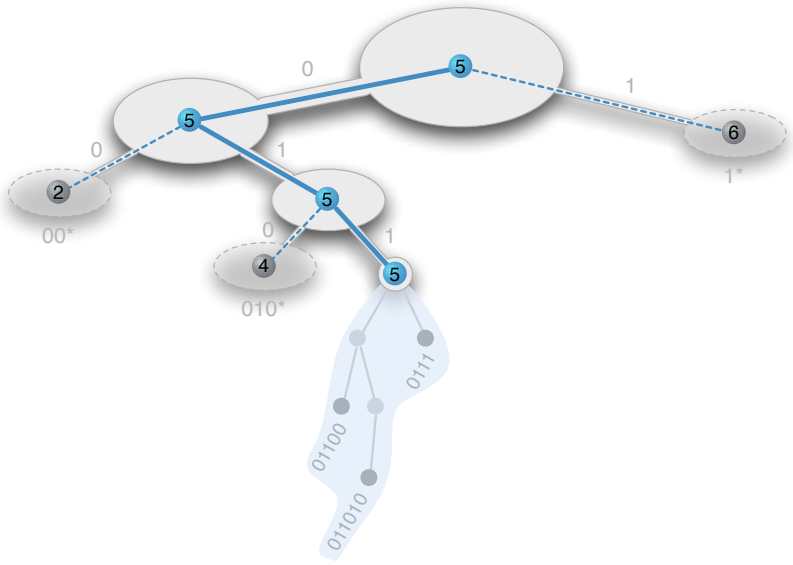


Figure 6: The local view of peer 5. Branch links are depicted by dashed lines.

during Pointer-Push&Pull operations. The latter guarantees that branch links point to truly random peers. Figure 6 shows the local view with trunk nodes and branch links of peer 5 (cf. Figure 3 for the global view).

Local branch links are similar to random branch links with the exception that these do not point to random peers of a subtree but to latency wise close peers of a subtree. Again Pointer-Push&Pull operations ensure that a peer sees all possible candidates for the local branch links over time and that the quality of the links improves quickly.

To join the network, a peer contacts an arbitrary peer p of the network and then proceeds as described in Algorithm 1. The joining peer will copy p's trunk node table for the root of the tree and then, based on the subtrees and weights given in the trunk node table, choose a subtree using the DHHT scheme. Using the list of branch links it is ensured that a peer p' that has chosen the same subtree can be contacted. Then, the same procedure is continued in the root node of the chosen subtree with peer p'. The algorithm terminates when either the joining peer is the only one assigned to a subtree or a leaf of the data tree has been reached.

Algorithm 1 Join(peer p)

1: initialize trunk node table for root node by copying p's trunk node table
2: $node \leftarrow$ root of tree
3: **while** number of peers in $node > 1$ and $node$ is not leaf of data tree **do**
4: $\quad node \leftarrow$ subtree calculated using DHHT
5: $\quad p' \leftarrow$ peer participating in $node$ found using branch link table
6: $\quad$ contact peer p' in $node$ and initialize trunk node table for $node$ by copying table of p'
7: **end while**

The subtrees chosen in line 4 of the join algorithm heavily depend on the correctness of branch link list and the weights obtained from the copied trunk node tables. Hence, it is crucial that all peers have consistent and accurate information about the structure and weight of the tree. Before we describe how this information is exchanged among peers let us recall the definition of weight used here. The weight $w(v)$ of a node v of

the network tree is given by the number of data items it stores[4]. The only peer allowed to set $w(v)$ is the peer responsible for v. The weight $w(T_v)$ of the subtree T_v rooted at v is determined by summing up $w(v)$ and the weights of the subtrees rooted at v's child nodes.

Peers exchange the information about tree structure and weights in the random networks of their trunk nodes. This is where Pointer-Push&Pull operations play an important role: whenever two peers communicate during a Pointer-Push&Pull operation their responsibility list, branch link list, and weights of the node and subtrees is piggy-backed to the messages. A peer p that has communicated with a peer p' uses the obtained information to update its own trunk node table as follows:

- The weight of the node is updated. As mentioned above, the only peer allowed to set the weight of a node is the responsible peer. Since the weight of a node may change over time each weight distributed by the responsible peer is tagged with a timestamp. Thus, peers will only update this entry in the trunk node table if they receive a weight with newer timestamp.
- Entries of the branch link list corresponding to subtrees peer p' actively participates in (i.e. subtrees p' has trunk nodes in) are set to point to p'. This way the entries of the branch link list are continuously replaced by peers which are ensured to be reachable. Furthermore, entries for subtrees not existent in the own branch link list are copied and entries that have been identified to be dead during Pointer-Push&Pull operations or routing are replaced. Most importantly the combination of update procedure and Pointer-Push&Pull operation guarantees that branch links point to truly random peers and that over time all peers participating in the trunk node are contacted.
- Entries of the responsibility list are replaced if peers with higher responsibility are found in the list of p' or added if the own list has less than r entries.

When changes to the weights or branch link list have been made a peer recalculates its own assignment to the subtrees and may change its path, i.e. trunk nodes, in the network tree when necessary.

The exchange of information using Pointer-Push&Pull operations is closely related to randomized rumor spreading using push and pull operations introduced by Karp et al. [24]. A major difference making a formal analysis difficult is that in our case the underlying communication network is not a complete graph and furthermore changes over time. Yet, if we assume the communication graph to be random we expect the dissemination of information by Pointer-Push&Pull operations to behave comparably as in [24], where $\mathcal{O}(n \ln \ln n)$ messages are sufficient to spread a rumor among n nodes w.h.p.

3.5 Routing

A lookup algorithm for the 3nuts network is given by Algorithm 2. The lookup is started at an arbitrary peer p and the only parameter is the identifier key of a data element. Since the data tree is a prefix tree defined by the data elements currently available in the network, the identifier key actually describes a path in the network tree. To reach the node of the network tree storing key, peer p follows the path key in its local view of the network tree until a leaf node is reached. This leaf node can either be a branch link or a trunk node in p's local view. In the former case the lookup is forwarded to the corresponding peer in the branch link list. In the latter case the lookup is forwarded to the peer responsible for the trunk node (since number of data elements is typically quite larger than the number of peers and most data elements reside in the leaves of the tree, it is very likely that the responsible peer is reached directly).

The following theorem gives a bound for the number of hops needed by the lookup operation to reach the peer responsible for a particular data element.

[4]Note that also internal nodes of the data tree may have load > 0, since we allow to store data on internal nodes.

Algorithm 2 Routing: Lookup(key) at peer p

```
1: if p has branch link to a peer p' sharing longer prefix with key then
2:     forward Lookup(key) to p'
3: else
4:     node ← last node of path key in local tree of p
5:     p' ← peer responsible for node
6:     if p = p' then
7:         return p
8:     else
9:         forward Lookup(key) to p'
10:    end if
11: end if
```

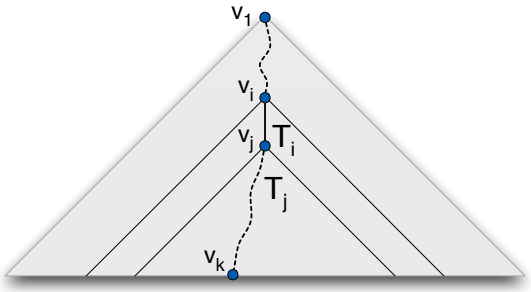


Figure 7: Partitioning of the network tree into subtrees as used in Theorem 3. The path of the lookup is given by $P = (v_1, \ldots, v_i, v_j, \ldots, v_k)$. T_i is the smallest subtree rooted on P with $|T_i| \geq \frac{n}{2}$.

Theorem 3 *In a 3nuts network with n peers the number of hops for a lookup operation is bounded by $\mathcal{O}(\log n)$ w.h.p., regardless of the structure of the data tree.*

Proof: To proof the theorem we will bound the number of hops needed to reach a subtree T containing at most half of the peers. Let $P = (v_1, \ldots, v_i, v_j, \ldots, v_k)$ be the path starting at the root of the network tree leading to the target node of the network tree. Furthermore let T_i and T_j be the subtrees rooted at v_i respectively v_j. We choose v_i and v_j such that $|T_i| \geq \frac{n}{2}$ and $|T_j| \leq \frac{n}{2}$. In other words: T_i is the smallest subtree rooted on P containing at least half of the peers.

The lookup starts at an arbitrary peer p in v_1, i.e. the root of the network tree. Let p' the peer reached by the first hop. Since $|T_i| \geq \frac{n}{2}$ and branch links point to truly random peers, p' will lie in T_i with probability at least $\frac{1}{2}$. If, on the other hand, p' does not lie in T_i, the same argumentation holds for the next hop from p', i.e.: the next hop from p' will lead to a peer in T_i with probability at least $\frac{1}{2}$. So, we reach T_i with one hop with probability $\geq \frac{1}{2}$, with two hops with probability $\geq 2^{-2}$, and with k hops with probability $\geq 2^{-k}$. Thus, we have

$$E[\#\text{hops to reach peer in } T_i] \leq \sum_{k=1}^{k=\frac{n}{2}-1} k2^{-k} \leq 2\,.$$

Since T_i is the *smallest* subtree with $|T_i| \geq n/2$, once the lookup reached a node in T_i one more hop is sufficient to reach the subtree T_j with $|T_j| \leq \frac{n}{2}$. Therefore, in expectation 3 hops are needed to halve the number of peers. Due to the recursive structure of 3nuts the same line of arguments as presented above does hold for subtree T_j. This implies that after $\log n$ iterations respectively an expected number of $3 \log n$ hops,

the lookup has reached v_k, i.e. a single peer.

It remains to show that $\mathcal{O}(\log n)$ hops are sufficient to reach v_k with high probability. We have seen that

$$\Pr[\text{\#peers is halved within three hops}] \geq \frac{1}{2} + \frac{1}{4} = \frac{3}{4} .$$

Dividing the lookup into sequences of three hops allows us to reduce the analysis to a sequence of mutually independent random variables $X_1, X_2, \ldots, X_{c \log n}$ taking values 0 and 1 with $\Pr[X_i = 1] = \frac{3}{4}$ respectively, and $X = \sum_{i=1}^{c \log n} X_i$. This allows us to apply Chernoff bounds and show that for $c \geq 5$ we have $X > \log n$ with high probability, i.e. the target peer is reached after $3c \log n$ hops with high probability. The expected number of successful steps is given by

$$E[X] = \frac{3}{4} c \log n .$$

Choosing $\delta = 1 - \frac{4}{3c}$ and applying Chernoff bounds we have

$$\begin{aligned} \Pr[X \leq \log n] = \Pr[X \leq (1-\delta)E[X]] &\leq e^{-\frac{1}{2}\left(1-\frac{4}{3c}\right)^2 E[X]} \\ &\leq e^{-\frac{1}{2}\left(\frac{11}{15}\right)^2 \frac{3}{4} c \log n} \\ &\leq n^{-\frac{121}{600}c} \\ &\leq n^{-c'} . \end{aligned}$$

So, the probability to be successful less than $\log n$ times is polynomially small in n. This implies that the lookup operation will need at most $3c \log n$ hops w.h.p. if we choose $c \geq 5$. Since we did not make any assumptions on the structure of the network tree this result does also hold for degenerated trees with linear depth. □

It is notably and important that the bound given by Theorem 3 holds regardless of the structure of the data tree since real world data will not be uniformly distributed but rather resemble a Zipf distribution. So, the data tree (and thus the network tree) will be unbalanced and exceed logarithmic depth. The reason that Theorem 3 holds for skewed data distributions is that branch links point to random peers of subtrees. Due to the properties of the Pointer-Push&Pull operation (see Theorem 2) and the way branch links are replaced (see Section 3.4) we can we can guarantee these to be truly random. This distinguishes 3nuts from other tree based peer-to-peer networks relying on heuristics mostly, e.g. P-Grid [2].

Algorithm 2 can be extended to perform range queries easily. To search for all data elements in a range $[x, y]$ the peer initiating the query calculates the longest common prefix z of x and y. Then, the query is routed to the node v of the network tree representing z. The subtree rooted at v is the smallest subtree containing all elements in the range $[x, y]$. Starting from v the query is forwarded to all child nodes holding data in the range $[x, y]$ in parallel until the leaf nodes of the network tree are reached.[5] Nodes receiving the query message will send their list of data elements to the peer that originated the query respectively forward the query to the responsible peer in case of internal nodes of the network tree.

4 Locality in 3nuts

Since the peers of a 3nuts network resemble the prefix tree defined by the data available in the network, closely related data elements are stored on network-wise close peers. In particular, the following theorem holds.

Theorem 4 *Let d be the distance of two data elements x and y in the tree metric. Then, x and y can be reached within d hops from one another in 3nuts.*

[5] Note that the border area of the subtree rooted at v may contain data elements not lying in $[x, y]$.

Proof: Let z denote the node of the network tree representing the longest common prefix of x and y. Note that w is present in the local tree of the peer responsible for x since z lies on the path starting at the root of the tree leading to x. So, no hops are needed to reach node z. From node z it is ensured that y can be reached within d hops since each hop will advance at least one level in the tree and the distance between z to y is bounded by d. The same line of arguments holds when routing from y to x. Recall that in any case the maximum hop distance is bounded by $\mathcal{O}(\log n)$ w.h.p., with n denoting the number of peers. □

Recalling the types of locality introduced in Section 2 this means that 3nuts provides information locality and in the previous section we have seen how range queries can be processed efficiently in 3nuts. We will now see how network and interest locality is provided by 3nuts.

3nuts provides network locality, i.e. lookups with small latency, through the list of local branch links maintained in the trunk node tables. As discussed in Section 3.4 local branch links point to a latency wise close peers of the corresponding subtrees and may be used for routing instead of random branch links. Initially the local branch link list is just a copy of the random branch link list. Latencies are measured during Pointer-Push&Pull operations and whenever a peer with lower latency is met it will be saved in the local branch link list. Since a peer will see every other peer of the random graph over time it is ensured that at some point the entries of the local branch link list point to the closest possible peers. In Section 5 we experimentally verify the quality of local branch links and compare latencies when routing with random and local branch links.

3nuts provides interest locality by allowing peers to *volunteer* for particular nodes of the data tree. Note that volunteering for the administration of a node does not relieve peers from participating in the regular peer assignment using DHHTs described in Section 3.2, i.e. volunteering is completely independent of the regular peer assignment and implies additional workload for a peer. However, volunteering allows a peer to dramatically decrease access times to parts of the data tree that are close to the node it is volunteering for. A peer p volunteering for a node v will actively participate in the path starting at the root of the data tree leading to v. For nodes of this path that are not coincident with the regular path p has been assigned to, p will have to maintain additional trunk nodes. Since the decision about responsibility is made using heights in the DHHT scheme (see Section 3.2), two special flags are used. The *volunteer* flag indicates that a peer volunteers to be responsible for this node and the *volunteer_down* flag indicates that a peer volunteers for a node further down the tree. The latter flag is used to prevent a volunteering peer to be responsible for nodes on the path to v, i.e. a peer with *volunteer_down* flag always has lower responsibility than peers regularly assigned to a node. In the node a that the peer is volunteering for, it sets the *volunteer* flag and therefore always has higher responsibility than peers that were assigned regularly to the node. In case of several peers volunteering for the same node the heights determined in the DHHT scheme are used to determine responsiblity.

5 Experimental Evaluation

To verify the robustness and practicability of 3nuts we used a prototype implementation in Java.[6] All experimental results presented in this section have been generated using this prototype. For the experiments the degree of random networks was set to $d = 3$. If not stated otherwise, the network consisted of $n = 2^{14}$ peers. Each peer stored five data elements giving a total of 81920 data elements. In most measurements several types of data have been used to verify the impacts of data distribution and degree of data respectively network tree. These are:

binary tree (uniform) A binary tree with data elements representing binary strings of length 40. The strings are chosen uniformly at random.

[6]The prototype is available for download at `http://3nuts.upb.de`

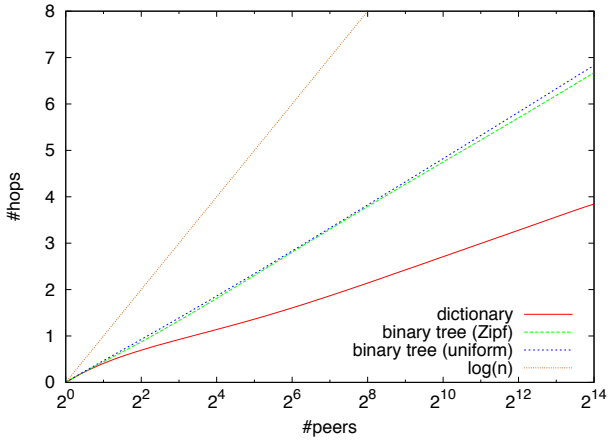


Figure 8: Average number of hops needed by the lookup operation.

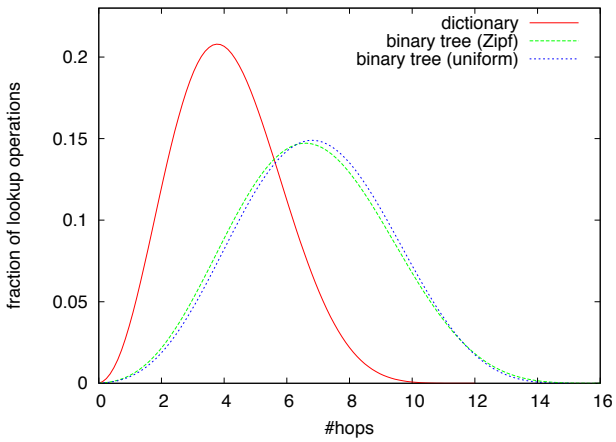


Figure 9: Distribution of number of hops needed by the lookup operation.

binary tree (Zipf) A binary tree of data elements representing binary strings of length 52, chosen according to a Zipf distribution as follows. Nodes in depth 20 of a complete binary tree have been assigned probabilities following a Zipf distribution with exponent set to 1. Data elements are placed by choosing a prefix of length 20 according to the assigned probabilities and concatenating a (unique) binary random string of length 32. So, some subtrees contain by far more data elements than others.

dictionary A tree generated by choosing data elements uniformly at random from a list of english words. Here, we used the freely available word list of the unix tool ispell [1]. In contrast to the binary trees, nodes of the dictionary tree have degree up to 26.

All of the following measurements have been performed multiple times and the curves represent the mean of these measurements.

5.1 Routing

Figure 8 shows the average number hops needed by the lookup operation for different data distributions and networks up to 2^{14} peers. A single measurement for a fixed network size and data distribution was done by performing 10^6 random lookup operations chosen from all possible combinations of peers and data elements. Here, the two curves representing the binary trees are almost equal, implying that the DHHT based load balancing performs excellent and the scalability of 3nuts is not affected by non-uniform data distributions. In case of the dictionary data distribution fewer hops are needed since data and network tree have substantially higher degree and thus the depth of the network tree is smaller than in case of the binary trees. All curves lie beneath the $\log n$ curve showing that the performance of the lookup operation is better than the performance formally proven in Theorem 3.

In Figure 9 the hop distribution measured in the network with 2^{14} peers is shown. The number of hops can vary between 0 (if the peer initiating the lookup itself is responsible for the data item) and the depth of the network tree. The latter is a merely theoretical bound since it is very unlikely to advance only one level of the tree with each hop of the lookup. However, greater depth of the data tree leads to higher variance of the hop distances. Nevertheless, the maximum hop distance measured in case of the binary trees is still bounded by $\log n$.

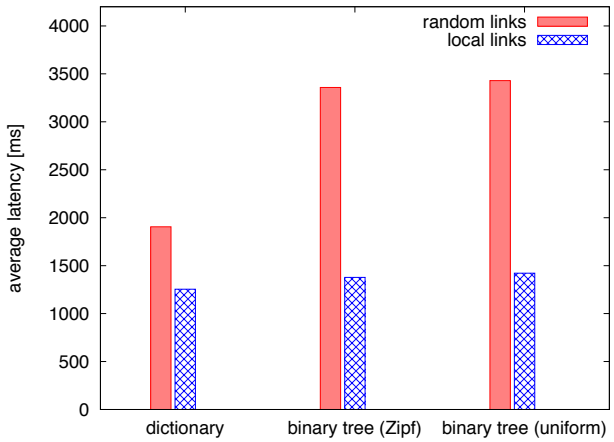


Figure 10: Average latency of the lookup operation: random branch links vs. local branch links.

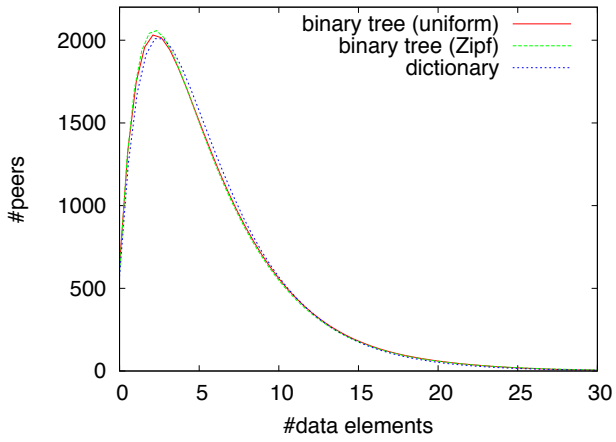


Figure 11: Load balancing: uniform distributed data, Zipf distributed data, and dictionary.

5.2 Network Locality

To evaluate the benefit of using local branch links instead of the random branch links we used the Georgia Tech Internetwork Topology Model [47] to model the physical network lying under the 3nuts overlay. Figure 10 shows the average latencies for different data distributions when using random respectively local branch links during the lookup operation. First of all, we observe that the average latency measured for routing with random links in binary trees exceeds the average latency measured for routing with random links in the dictionary tree by a factor of 1.8. This is explained by the larger number of hops needed in the comparatively deep binary trees. When using local branch links latencies are reduced significantly for all three types of data trees. Notably, average latencies measured for dictionary and binary trees then only differ by a factor of 1.1, i.e. the binary trees benefit more from the use of local links. This is explained by the fact that the last few hops during a lookup are by far the most "expensive" ones during a lookup operation since the number of peers to choose from decreases with each hop, i.e. it is more unlikely that latency wise close peers are among the peers to choose from. Thus, the last hops makes up a large fraction of the total latency during a lookup.

5.3 Load Balancing and Degree

Figure 11 shows the distribution of data load among peers. Each peer contributes 5 data elements and thus the average load is also 5. Since the load balancing using DHHT is based on randomization there are some peers exceeding the average load by factor 5. Anyhow, 90% of the peers have at most twice the average load. While having peers exceeding the average load by factor 5 is for sure not optimal, one has to recall the simplicity and elegance of the load balancing scheme used here. It is in particular remarkable that the decisions a peer makes when choosing its path are completely independent of decisions made by other peers. So, a peer usually does not have to change its path respectively responsibility when further peers enter the network. Coping with not 100% fair load balancing is the price to pay for this simplicity. Note however that simplicity may be crucial to keep the network stable under churn. Furthermore, a peer noticing its load exceeding the average by a large factor still has the option to leave and rejoin the network. Due to the randomized nature of the load balancing scheme used here, it will most likely be assigned to a different part of the tree.

Another interesting measure is the number of links, i.e. neighbors, a peer has to maintain. Figure 12 shows the sum of branch links and neighbors in the random networks per peer while Figure 13 shows the

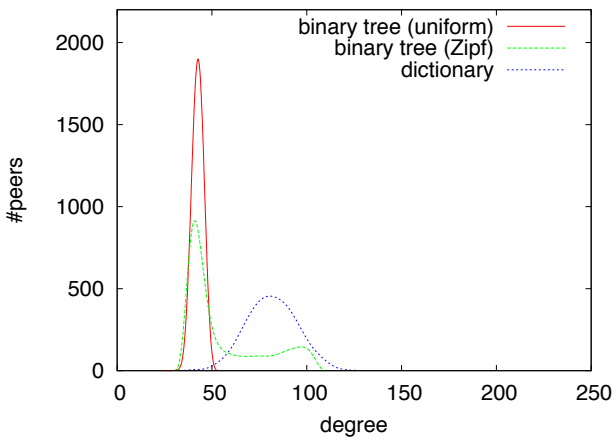


Figure 12: Degree distribution: number of branch links and random neighbors per peer.

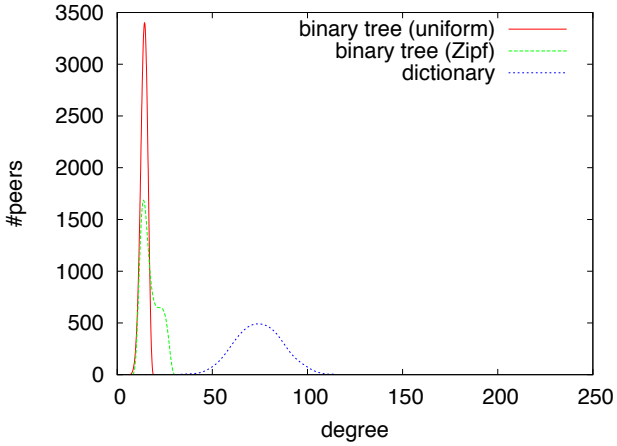


Figure 13: Degree distribution: number of branch links per peer.

number of branch links per peer only. The degree of a peer of course depends on the degree and the depth of the data tree: high degree involves a large number of branch links and higher depth involves a larger number of trunk nodes and thus random networks a peer is participating in. We start by analyzing the uniformly distributed binary tree. Considering the number of branch links only (Figure 13), a peers degree is comparable to the degree of a peer of a standard DHT based network like Chord [45], i.e. in case of the uniformly distributed binary tree the average degree is 14.5 and the expected degree in a plain Chord network of size 2^{14} is 15. Taking links in random networks into account (Figure 12), the average degree increases to 42.

Zipf distributed data leads to slightly increased average degree compared to the uniformly distributed binary tree. This is explained by the fact that data and network tree have greater depth. In case of the dictionary tree the high fan out of the data tree imposes a large number of branch links to the peers. A solution to reduce the number of branch links here is to convert the tree into a binary tree, i.e. interpret the characters of data elements bit wise instead of byte wise. Furthermore, the depth of the tree and thus the average peer degree can be reduced by using radix trees [15] instead of a simple prefix tree. A radix tree, also known as Patricia trie/tree, or crit bit tree, is a specialized form of prefix tree. In contrast to a regular prefix tree, the edges of a radix tree are labelled with sequences of characters rather than with single characters. This allows to reduce the depth of the tree if sub-paths without branches exist in the tree, what is likely when using bit wise interpretation of characters.

Considering the additional features of 3nuts compared to DHT based networks, e.g. information locality and network locality, the "costs" for these features, i.e. the increased degree, is negligible. Moreover, links in 3nuts are very easy to maintain: While links in most peer-to-peer networks have to point to one particular peer, a link in 3nuts may always point to a random peer out of a large set of candidates.[7] This is especially true for the random networks where even multi-links are allowed so that a failing neighbor can simply be replaced by duplicating another link — Pointer-Push&Pull operations ensure the network structure to be re-randomized after only a short time.

5.4 Dynamics and Robustness

We also verified the robustness of 3nuts when a substantial fraction of the peers fails unexpectedly. A network with 10^4 peers was generated and once the network stabilized, 25% of the peers were removed

[7] With exception of the lowest levels of the network tree.

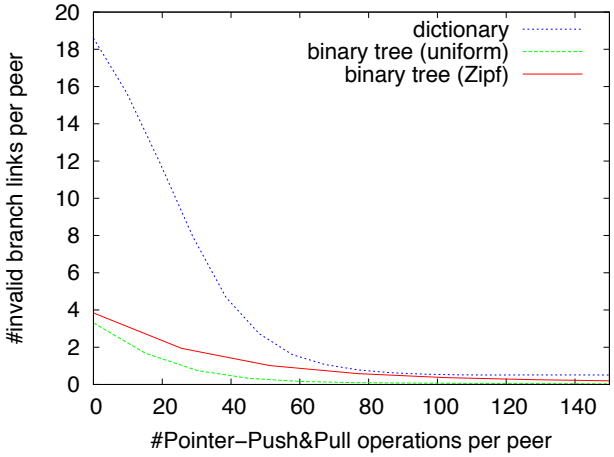


Figure 14: Evolution of invalid branch links after failure of 25% of peers.

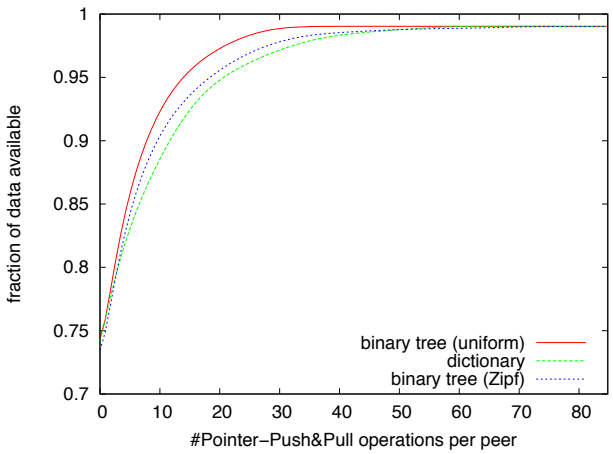


Figure 15: Evolution of data availability after failure of 25% of peers.

simultaneously. Figure 14 shows the evolution of the average number of invalid random branch links per peer (local branch links were neglected here). Due to the higher fan-out the number of branch links is significantly higher in case of the dictionary tree and thus the number of invalid links is also higher compared to the binary trees. The axis of abscissae shows the total number of Pointer-Push&Pull operations per peer, i.e. cumulative for all random networks a peer participates in. In case of the binary trees about 20 random Pointer-Push&Pull operations are sufficient to reduce the number of invalid links per peer to 2 while this takes about 50 operations in case of the dictionary tree. Here, the comparably strong decrease during the first 50 operations is explained by the randomness of Pointer-Push&Pull operations and the larger number of invalid links, i.e. links are "checked" randomly and the larger the number of invalid links, the higher the probability to find these. Note however that the dictionary tree may be converted to a binary tree as mentioned above. Furthermore, when encountering an invalid branch link during a lookup operation, the lookup can simply be forwarded to a random neighbor in the trunk node. Since branch links are ensured to be random the branch link list of this neighbor most likely has a different link to the designated subtree.

Figure 15 shows the evolution of data availability in the same scenario. Data availability was checked by performing 10^6 random lookup operations respectively. Here, a lookup was considered as failed whenever an invalid branch link was encountered and furthermore the repair process usually initiated when invalid links are encountered during a lookup was disabled. In other words: Trunk node tables were only updated/repaired by Pointer-Push&Pull operations. The different data distributions respectively tree types behave almost equal. Right after the removal of peers about 75% of the data remaining in the network is still available and after 30 to 50 Pointer-Push&Pull operations per peer data availability of 99% is reached. We stopped the experiment at 99% availability since the removal of 2500 peers and thus 12500 data elements also involves changes in the network tree and some peers change their path, etc.

6 Concluding Remarks

The 3nuts peer-to-peer network introduced here combines unstructured and structured peer-to-peer networking concepts. To allow topological search, 3nuts combines random networks, prefix trees, and distributed hash tables. To the best of our knowledge 3nuts is the first peer-to-peer network providing interest, network and information locality at the same time. The practicability of 3nuts has been affirmed by a prototypical implementation ready for practical use, experimental evaluation, and verification on a mathematical level.

The 3nuts architecture has been designed around the Pointer-Push&Pull operation, whose properties make it an excellent maintenance operation for dynamic networks. Replacing the heartbeat (ping) messages between peers, Pointer-Push&Pull is used in 3nuts to:

- maintain truly random networks to replace nodes of the data tree and thus make the network robust,
- spread information among peers and thus give peers a coherent view of the tree,
- maintain branch links and guarantee them to be truly random, thus allow efficient routing,
- measure round trip times (RTT) and thus allow routing with small latency.

A possible drawback of the 3nuts architecture is the potentially high degree of the network, which can be caused by highly skewed data distributions resulting in deep paths. However, as pointed out in Section 5.3 one should keep in mind that maintenance of links is comparably cheap in 3nuts and higher degree also implies higher robustness. If necessary, the degree can possibly be reduced by making use of radix trees or balanced trees instead of simple prefix trees.

References

[1] International ispell. http://www.lasr.cs.ucla.edu/geoff/ispell.html.

[2] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: a self-organizing structured p2p system. *SIGMOD Record*, 32(3):29–33, 2003.

[3] James Aspnes and Gauri Shah. Skip graphs. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003.

[4] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[5] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Computer Communication Review*, 34(4):353–366, 2004.

[6] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407–418, New York, NY, USA, 2003. ACM.

[7] Clip2. The Gnutella protocol specification v0.4. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, 2001.

[8] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 25–30, New York, NY, USA, 2004. ACM.

[9] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *1st Symposium on Networked Systems Design and Implementation*, pages 85–98, March 2004.

[10] Cedric du Mouza, Witold Litwin, and Philippe Rigaux. SD-Rtree: A scalable distributed rtree. In *Proceddings of the 23rd International Conference on Data Engineering (ICDE 2007)*, pages 296–305, April 2007.

[11] Pierre Fraigniaud and Philippe Gauron. D2b: a de bruijn based content-addressable network. *Theoretical Computer Science*, 355(1):65–79, 2006.

[12] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 444–455. VLDB Endowment, 2004.

[13] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. *ACM SIGOPS Operating Systems Review*, 37(5):314–329, 2003.

[14] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, New York, NY, USA, 1984. ACM.

[15] Gernot Gwehenberger. Anwendung einer binären verweiskettenmethode beim aufbau von listen. *Elektronische Rechenanlagen*, 10(5):223–226, 1968.

[16] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peer-to-peer networks. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 242–259, London, UK, 2002. Springer-Verlag.

[17] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[18] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distribted object location in a dynamic network. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA-02)*, pages 41–52, New York, August 10–13 2002. ACM Press.

[19] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *VLDB '2003: Proceedings of the 29th International Conference on Very Large Data Bases*, pages 321–332. VLDB Endowment, 2003.

[20] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 661–672. VLDB Endowment, 2005.

[21] H. V. Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 34, Washington, DC, USA, 2006. IEEE Computer Society.

[22] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, pages 98–107, Berkeley, California, 2003.

[23] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, 4–6 May 1997.

[24] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. Randomized rumor spreading. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 565, Washington, DC, USA, 2000. IEEE Computer Society.

[25] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating dht design tradeoffs under churn. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, pages 225–236, March 2005.

[26] Mei Li, Wang-chien Lee, and Anand Sivasubramaniam. DPTree: A balanced tree based indexing framework for peer-to-peer systems. In *ICNP '06: Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 12–21, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Chu Yee Liau, Wee Siong Ng, Yanfeng Shu, Kian-Lee Tan, and Stéphane Bressan. Efficient range queries and fast lookup services for scalable p2p networks. In *DBISP2P*, Lecture Notes in Computer Science. Springer, 2004.

[28] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th International Conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[29] Peter Mahlmann and Christian Schindelhauer. Peer-to-peer networks based on random transformations of connected regular undirected graphs. In *SPAA '05: Proceedings of the seventeenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 155–164, New York, NY, USA, 2005. ACM.

[30] Peter Mahlmann and Christian Schindelhauer. Distributed random digraph transformations for peer-to-peer networks. In *SPAA '06: Proceedings of the eighteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 308–317, New York, NY, USA, 2006. ACM.

[31] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *PODC '02: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 183–192, New York, NY, USA, 2002. ACM Press.

[32] Anirban Mondal, Yi Lifu, and Masaru Kitsuregawa. P2PR-Tree: An r-tree-based spatial index for peer-to-peer environments. In *Current Trends in Database Technology - EDBT 2004 Workshops*, pages 516–525, 2004.

[33] Alberto Montresor, Mark Jelasity, and Ozalp Babaoglu. Chord on demand. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 87–94, Washington, DC, USA, 2005. IEEE Computer Society.

[34] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *SPAA '03: Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 50–59, New York, NY, USA, 2003. ACM Press.

[35] Bill Pickelhaupt and Kevin Starr. *Shanghaied in San Francisco*. Mystic Seaport Museum, 1970.

[36] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa. Accessing nearby copies of replicated objects in a distributed environment. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 311–320, New York, June 1997. ACM Press.

[37] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[38] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief announcement: prefix hash tree. In *PODC '04: Proceedings of the 23rd annual ACM Symposium on Principles of Distributed Computing*, pages 368–368, New York, NY, USA, 2004. ACM.

[39] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. Technical report, Intel Research Berkeley, 2004.

[40] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[41] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: search methods. *Computer Networks*, 50(17):3485–3521, 2006.

[42] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[43] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN'02)*, 2002.

[44] Christian Schindelhauer and Gunnar Schomaker. Weighted distributed hash tables. In *SPAA '05: Proceedings of the seventeenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 218–227, New York, NY, USA, 2005. ACM.

[45] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In Roch Guerin, editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, August 27–31 2001. ACM Press.

[46] D. A. Tran and T. Nguyen. Hierarchical multidimensional search in peer-to-peer networks. *Computer Communications*, 31(2):346–357, 2008.

[47] Ellen Zegura, Kenneth Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *IEEE INFOCOM '96: Proceedings of Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation*, volume 2, pages 594–602, 1996.