FPN: A Distributed Hash Table for Commercial Applications

Cezary Dubnicki, Cristian Ungureanu, Wojciech Kilian
NEC Laboratories
Princeton, NJ, USA
{dubnicki, cristian, wkilian}@nec-labs.com

Abstract

Distributed hash tables (DHTs) provide a scalable mechanism of mapping keys onto values. DHTs are designed for fully decentralized, yet efficient object location in peer-to-peer systems. The ad-hoc and dynamic nature of P2P networks motivated existing DHTs to keep only minimum state per node, resulting in relatively long routing paths. Moreover, since the storage in existing P2P systems is essentially "free", its utilization has not been the primary focus of DHT design, resulting in systems with poor utilization.

This paper presents Fixed Prefix Network *(FPN), a prefix-based DHT designed for future commercial P2P systems supporting applications like distributed archive repository and distributed DNS. Unlike traditional P2P, the new breed is built on the assumption that the contributed resources are dedicated to the system, and are significantly more stable. Exploiting this characterization, FPN allows trading of state size for routing length, making it possible to maintain short fixed path lengths for a wide range of number of nodes. Moreover, FPN guarantees the minimum storage utilization, and in practice can deliver an 80% utilization during the lifetime of a growing system. Finally, FPN is based on a simple concept, yet delivers scalability and robustness similar to other DHTs.*

1. Introduction

Distributed hash tables, or DHTs, are designed for application in P2P systems to locate objects without the need for a centralized server. Nearly all recent DHT proposals [13, 9, 4] have a lot in common - they typically use a small routing table per node, and can locate any object in a number of steps growing with the size of the system. The routing tables connect the nodes in an overlay network, with multiple paths between any two nodes which ensures not only scalability, but also robustness in the face of frequent topology changes like node additions or failures.

While P2P file sharing systems are very successful, we envision the application of some of the principles behind these systems to the commercial domain, in particular to the enterprise and service sectors. Features like scalability, dynamic topology, automatic failure recovery, and minimal requirement for human management are all very attractive in the business world as they lead to increased system reliability, increased system lifetime and reduced total cost of ownership. Such systems, which we call *ComP2P* for commercial P2P, consist of a large-scale network built of dedicated resources. A primary example of such a system is a distributed archival repository. Like traditional P2P, these systems are fully decentralized, and are robust in the face of failures. However, the resources are not shared in an ad-hoc and transient way like in traditional P2P, and cannot be removed from the system on a whim. Instead, the sharing is long-term and resources are contributed in a well-defined manner. The management of these new systems is mostly local, but driven by a set of global parameters and rules. Local adherence to these rules guarantees global properties of the system. An example is that every local administrator must ensure that the storage space added locally is at least equal to the expected size of the data to be inserted locally divided by the global minimum utilization.

By taking full advantage of stable and dedicated resources *ComP2P* should be able to offer better performance than traditional P2P systems. The small state per node, which is either constant [6] or grows slowly with the network size [13, 9], supports a high rate of change in existing DHTs, but at a cost of relatively long paths. For example, a recent study [2] investigating the applicability of a traditional DHT to a reimplementation of a large distributed service like DNS reached a negative conclusion mainly because the DHT latency was order of magnitude higher than that of the current DNS.

Additionally, the concept of storage utilization is not crucial for the traditional P2P systems where contributed resources are essentially "free". However, ensuring the efficient usage of underlying resources is critical for the commercial viability of *ComP2P* systems.

This work introduces a new prefix-based DHT designed for *ComP2P* systems. The network is prefix-based because each overlay node is assigned a prefix of the hashkey space. Unlike previously published work in the DHT area, our approach is focused on exploiting stability and dedication of underlying resources to provide (1) shorter or even fixed-length routing paths across a wide range of number of nodes, and (2) good storage load balance and guaranteed storage utilization, while ensuring traditional DHT features like scalability, availability, and robustness.

2. FPN Definition

Fixed Prefix Network (FPN in short) consists of multiple *FPN nodes* delivering DHT functionality.

System Model The system consists of multiple physical nodes (*pnodes*) communicating through an underlying transport network. Pnodes can vary in storage space, processing power and other resources. Each pnode has a number of storage *slots* proportional to the amount of its storage. Each pnode may host multiple FPN nodes simultaneously up to its number of slots. There is a global limit on how many keys any FPN node can keep given by the constant *SlotSize*.

Zones and Prefixes Hashkeys can be viewed as fixed-size strings of bits in the space covering all possible combinations of such strings. A *zone* is a subspace of the hashkey space described by a prefix. If the hashkey space is spanned over S bits, then a given K-bit prefix ($K \leq S$)

$$f_1 f_2 \cdots f_k$$

(where f_i is a specific bit) describes a zone that keeps all objects with hashkeys starting with this prefix, *i.e.* of the form

$$f_1 f_2 \cdots f_k x_{k+1} x_{k+2} \cdots x_S$$

where x denotes a variable bit.

An FPN node is responsible for the zone represented by this prefix, which is also this node's id. We use "zone" and "FPN node" interchangeably except clearly marked cases when a zone not related to any FPN node is discussed.

Figure 1 illustrates zone prefixes. Two blocks with keys $K_1 = 00100101$ and $K_2 = 01011011$ would be stored on zones C and E, respectively. FPN nodes occupy only leafs of the prefix tree. The shorter the prefix, the bigger the zone represented by this prefix.

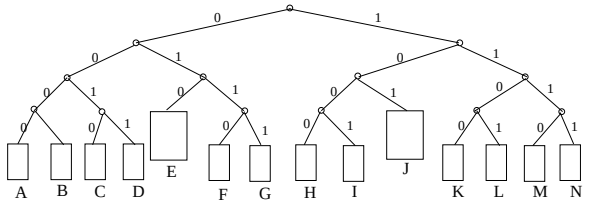

Figure 1. FPN example.

Neighbors Two prefixes F and G are said to be *neighbors along bit i* iff both the following conditions hold:

$$f_i = \bar{g}_i$$
$$f_j = g_j \quad \text{for } j \neq i \ \wedge \ 1 \leq j \leq \min(\text{len}(F), \text{len}(G))$$

In other words, the ith bit in F is flipped in G and all other bits are equal for bit positions occurring in both prefixes. Two prefixes are *neighbors* iff a bit position exists along which these prefixes are neighbors according to the above definition. Two zones are neighbors iff their prefixes are neighbors.

We say an FPN is in a *stable state* iff any two FPN nodes have non-overlapping prefixes and the entire hashkey space is covered by zones with assigned nodes.

Short prefix zones have more neighbors than long prefix zones. In Figure 1 J has 6 neighbors (C, D, H, I, M, N), while A has only 4 (B, C, E, H). This imbalance can be limited by splitting the bigger zones first. When an FPN network is perfectly balanced its topology is that of a binary hypercube.

3. FPN Operations

Some FPN operations, like locating overlay network when a new pnode is added, failure detection with pings, and failure recovery, can be done in similar fashion to CAN [9]. Below we concentrate on operations differentiating FPN from other DHTs.

Zone Split The split consists of creating two "children" zones out of a parent zone with prefixes derived from the parent prefix by appending 0 and 1 respectively. The parent's neighbors are split between the children, with some neighbors possibly retained by both. This operation is local: the children zones replace the parent on the node hosting it. Usually, a zone is split when its size reaches *SlotSize*; sometimes split may also be necessary during node addition.

Zone Transfer A zone is evicted when the storage on a pnode is exhausted or when there is no free slot for it locally. A pnode can become a new host for a zone F only

if it has at least one free slot; also, to prevent thrashing, its free space after the transfer must be higher than the source's before the transfer.

Search for Transfer Target Among candidates for transfer destination we prefer the one with the highest amount of free space. However, the "best" candidate can only be found by checking every pnode in the system (*i.e.* using *global knowledge*). Since this is impractical, we have implemented a *local scheme* in which every pnode maintains a set of candidates (of limited cardinality), called *transfer set* containing each candidate's IP and an estimation of its free resources – slots and storage space. This information about a node's free resources is piggybacked on outgoing messages. In our experiments we have used, besides pings, write messages because their number is proportional to the rate of growth (hence transfer) of the system.

Limiting the size of the transfer set not only bounds the space consumption at each pnode, but also allows to discard stale information.

4. Routing

4.1. Basic Routing

Basic routing is done by resolving bits left-to-right, which results in a number of hops on the order of logarithm of number of zones. For large networks this leads to big latencies. To reduce number of hops, we introduce the concept of jump tables.

4.2. Dimensions and Jump Tables

Besides regular neighbor tables, each zone keeps so-called *jump tables*, one in each dynamically defined *dimension*. Jump tables reduce the number of hops and improve resiliency.

A dimension consists of a fixed number of bits starting at a given hashkey bit position - for example it can be defined by a bit position 4 and length 3. A jump table for a given dimension has an entry for each possible combination of bits in that digit (and is indexed by that combination). Each entry is associated with a prefix, which is the identifier of the local FPN node with the dimension's bits replaced by this entry's index. The value of the entry is the set of FPN nodes responsible for entire hashkey space specified by this entry's prefix. Each such FPN node is described by its zone prefix, the host network address and the zone version, which is incremented when the zone is recovered or moved to a different host. The version is used in the jump table construction described below.

Entry Prefix Example Consider a zone with prefix 011011**00**101 and the 3-bit dimension which is in bold. The prefix associated with entry defined by index 5 is defined as 011011**10**101, *i.e.* the dimension bits (in bold) replaced with the binary representation of 5 (101). For a dimension consisting of 3 bits, its jump table has $2^3 = 8$ entries, each with a different index ranging from 0 to 7.

Jump Table Example Consider again the network in Figure 1 and the two bit dimension starting at the first bit. The jump table for zone E (with prefix **01**0) in this dimension has the nodes responsible for the following prefixes:

Index	Prefix	Destination Zones
0	**00**0	**00**00 and **00**01 (nodes A and B)
1	**01**0	**01**0 (this node)
2	**10**0	**10**00 and **10**01 (nodes H and I)
3	**11**0	**11**00 and **11**01 (nodes K and L)

Jump Table Construction Changes to a jump table in a bit dimension are propagated with pings to neighbors along bits of this dimension. A receiving zone updates its jump tables as part of the state reconciliation procedure using zone versions to select the latest destination zone location. If B is the number of dimension bits, full jump tables are constructed in B ping cycles on all zones involved.

If d is the number of dimensions, each with the same number of bits, and n is the number of zones, then the total number of destinations in jump tables of one zone is $O(dn^{1/d})$ provided the difference between the longest and the shortest prefix in the network is limited by a constant.

4.3. Routing with Jump Tables

Consider a prefix P and a hashkey K. Let $K|_P$ be the prefix of K with the same number of bits as in P. We define *LexDist*, the lexicographical order distance between P and K, to be the bit string given by:

$$LexDist(P, K) = \mathrm{XOR}(P, K|_P)$$

Note that $LexDist(P, K)$ contains all zeros iff prefix P includes hashkey K; at a given moment there is only one such prefix in a stable FPN network. We say that a prefix P_1 reduces *LexDist* to a given hashkey K compared to prefix P_2 iff $LexDist(P_1, K)$ is before $LexDist(P_2, K)$ in the standard lexicographical order.

Routing with jump tables tries to reduce *LexDist* to the destination by attempting complete digit resolution first, and partial digit resolution second. If this is not possible, the routing is done by resolving one bit

using neighborhood tables. In a network without failures this algorithm uses the first method only; bit resolution is used only for one hop when needed.

If all zones use the same dimension size, the maximum routing length is limited by the number of dimensions in the longest prefix provided there are no failures and jump tables are up-to-date.

4.4. Change of Dimension Size

FPN provides a mechanism to increase the dimension size on-line. It can be used to keep the number of dimensions (and indirectly the path length) constant. As nodes are added to the system, a split may cause the creation of a zone whose prefix is longer than the number of dimensions times the dimension size (in bits). When this happens, the dimension size is increased. The zone re-arranges its jump tables according to the new digit size (with some entries initially empty), and communicates the new size to neighbors through pings. In response, the neighbors make their jump tables bigger, and further propagate the change with pings.

Eventually, every node changes its dimension size, so in this sense it is a global operation. However, this change is done lazily while the system continues normal operation. Moreover, each node changes its dimension size only once in each period when the overall system size increases by a factor of 2^d where d is the number of dimensions.

The dynamic change of dimension size contributes to the efficient use of resources and overall system longevity. This not only allows FPN to avoid preallocating resources based on some maximum size network, but also allows it to work in case the system happens to grow over that size.

4.5. Routing Experiments

We have simulated an FPN built of pnodes with a capacity of 1000 keys each. The system tries to maintain routing path length equal to 3, *i.e.* the number of digits is 3 with each digit dynamically increased as the system grows.

We have simulated two scenarios.

- A static network where each node generates reads only.
- A growing network. with equal number of reads and writes generated per node. The writes fill up the nodes causing the system to grow until it reaches the maximum number of nodes shown. The results are cumulative up to each number of nodes.

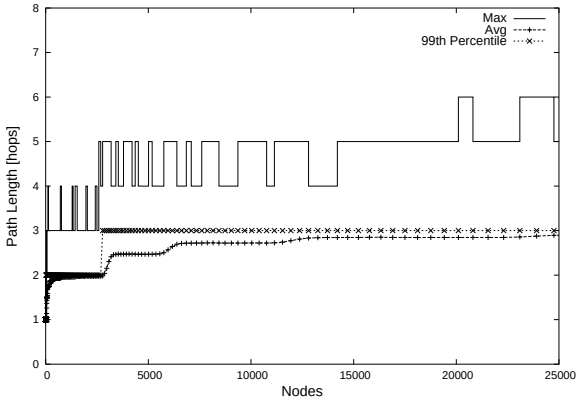

Figure 2. Path length in a growing network.

Figure 2 shows path lengths in a growing network with no failures. In this case the 99% of messages are delivered in no more than 3 hops.

To investigate the effects of node failures on routing we failed simultaneously a fraction of nodes from 10% up to 50% in steps of 10% in a static network without recovery.

Immediately after the one-time failure of a given fraction of nodes there is a stabilization period in which information about failures is propagated with pings.

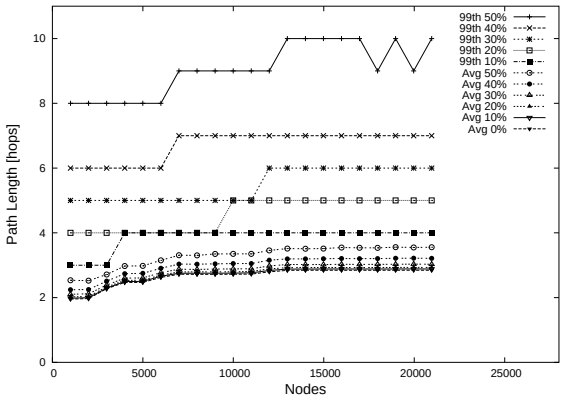

Figure 3. Average and 99^{th} percentile path length in network with failures during stabilization. The failure rate varies from 0% to 50%.

Figure 3 presents the behavior of a stabilizing network. The fraction of nodes failed has limited impact on the average path length. For a system of $20,000$ nodes, the average path increased by about 25% (from 2.84 to 3.55) when 50% of nodes failed. The 99^{th} per-

centile path length grows with the percentage of nodes failed reaching 10 when half of the nodes failed. In case of a stabilized network (not shown here) the average path length is similar but slightly shorter (increasing to 3.48), with the 99^{th} percentile path length reaching 9 after half the nodes failed.

5. Storage Utilization and Load Balancing

A system using a given load balancing scheme reaches *system full* state when insertion of the next hashkey cannot be performed due to the lack of space. We say that the *guaranteed utilization* is U if a system cannot be full if its utilization is below U. Note that the guaranteed utilization differs from the utilization at a given time; the latter can be lower than the former if too few keys were inserted.

We propose two techniques, *lazy splitting* and *oversubscription*. Used together, they can guarantee a utilization on the order of 80%. This guarantee not only holds for the system's entire lifetime (as it grows), but is also achievable with only a modest increase in bandwidth consumption.

Base scheme A simple way to add a new pnode is to pick a random key, determine the zone responsible for it, split this zone into two new zones, and transfer one of them to the new pnode. This scheme suffers from one big problem: randomly choosing the zone to split results in big inequalities in zones' areas (factors of $log(n)$ are possible, where n is the number of nodes in the system). Thus, when we reach system full state, some nodes will be only $1/log(n)$ full, leading to poor overall utilization.

Aggregation An improvement to the base scheme groups randomly selected zones into sets (of some fixed cardinality N), with pnodes hosting entire sets of zones (as opposed to only one zone). For example, a pnode with a 1TB capacity can host up to five zones that can grow to 200GB each, giving a value of $N = 5$. In this approach, a pnode has a number of *slots* of capacity given by the system-wide constant *SlotSize*. A zone growing beyond this limit must be split.

Because the likelihood that many of the bigger zones are grouped together is small, the ratio of the biggest set load to that of the smallest set load is significantly smaller than the ratio for individual zones.

Slots are similar to the *virtual nodes* used in load balancing in a number of systems [13, 9], but the virtual nodes do not have a capacity limit themselves, the only limit being that of the pnode. In [13] each pnode hosts a number of virtual nodes approximately equal to the logarithm of the number of nodes. Using this technique, the utilization is between 50% and 60% for a 10000 node system.

5.1. Lazy Splitting

This technique splits a zone only when it becomes full. For a given number of keys, it results in the minimum number of zones possible. Nodes added to the system before a split is needed constitute a *free slot reserve*, from which slots are used when zones are split.

Lazy splitting alone can bring the guaranteed system utilization to 50%, since it is easy to see that when the system becomes full the free slot reserve must be empty and each node is at least 50% full. Unfortunately, this is also a tight bound (a system can have all nodes but one half-full, and one node full).

To maintain the free slot reserve, FPN arranges that every node in the system hosts at least one FPN zone. When a new pnode is added, a zone from a "donor" pnode hosting two (or more) zones is transferred there; from such pnodes, the one with the highest number of zones is preferred in order to limit the imbalance. The search for a donor is not global; only up to a fixed number of pnodes are contacted.

If the system is lightly loaded, it is possible that no pnodes with two zones can be found, in which case we allow an *eager split* of the biggest zone found. It can be shown that the bound on the guaranteed utilization introduced in the next section does not change if each pnode can have at most one eagerly split zone; having two eagerly split zones is clearly not necessary since the same zone can be eagerly split many times (each time one of new zones is transferred to a new pnode). Once an eagerly split zone becomes full, it reverts to being a 'normal' zone.

5.2. Oversubscription

To achieve utilization higher that 50%, each pnode can hold more slots than would actually fit given its capacity (*i.e.* pretend to have more space than it's actually available). The key observation is that, if the zones residing on a node are lightly loaded (as most of them are when the utilization is low), there is enough storage on the node to host extra zones. Consequently, we modify the aggregation technique so that a node has $2 \times N - 1$ slots, where $N = C/SlotSize$, and C is the node's capacity. When the pnode becomes full before any of its zones is full we perform a *transfer* as described in Section 3.

It can be shown that the lazy splitting with the oversubscription can guarantee utilization $(N - 1)/N$.

For $N = 5$ this results in a guaranteed utilization of 80%. This scheme can be easily applied to heterogeneous networks by assigning to each pnode a number of slots equal to $(2 \times N_{node} - 1)$ where $N_{node} = \lfloor C_{node}/SlotSize \rfloor$. The guaranteed utilization for such system is given by $(\overline{N} - 1)/\overline{N}$, where $\overline{N}$ denotes the average (over the entire network) number of slots per node.

Since the guaranteed utilization is independent of the system size, a credit-debit system can be run locally: after a given amount of storage is contributed locally, new data can be inserted there up to the size equal to the guaranteed utilization times the size of the contributed storage. This not only allows local administration of the system, but also reduces the bandwidth requirements compared to adding nodes only when system is full.

5.3. Utilization Experiments

In these experiments we subjected the system to hashkey insertion (write) and measured the utilization and the *transfer rate* obtained by various load balancing techniques. The transfer rate is defined as the ratio of the number of key transfers to the number of keys stored in the system. The transfer number includes all hashkey moves due to either insertion (write) or zone move since the start of the system.

While the order in which nodes or keys were added does not influence the utilization, it has a big impact on transfer rate. At one extreme, all the nodes are added to the system before any key is inserted. Since splits and transfers executed eagerly by empty zones do not move any data, the only data transfer is caused by (subsequent) write operations, thus providing the best-case scenario. Each writes generates one transfer; thus, the lowest possible transfer rate is 100%. The other extreme is when pnodes are added only when the system becomes full. This forces transfer zones from pnodes that filled up to pnodes with enough free space.

Benefits of Oversubscription Figure 4 shows the utilization and worst-case transfer rate with and without oversubscription as a function of the number of nodes in a system when $N = 4$ for each node. In such case, the theoretically proved guaranteed utilization for oversubscription is 75%, and the measured utilization minimum is 85%.

Disadvantages of Oversubscription Increasing the number of zones per node leads to a larger fraction of node's storage dedicated to various structures (such as neighborhood and jump tables) needed for each zone. This problem is shared by the simple aggregation scheme.

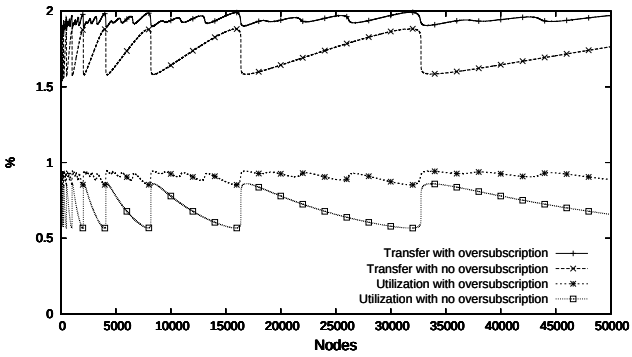

Figure 4. Utilization and transfer rate for $N = 4$

Figure 4 shows another cost - extra traffic caused by evicting zones when nodes become full. Compared to the base case without oversubscription, the average transfer rate increases by about 12% from 1.75 to 1.95. The next section describes *oversubscription with threshold*, a technique providing the same guaranteed utilization with lower bandwidth requirements. Note that oversubscription does not have a "bandwidth penalty" for read operations; thus, in a system with 80% reads and 20% writes, the increase in bandwidth consumption is about 2.4%.

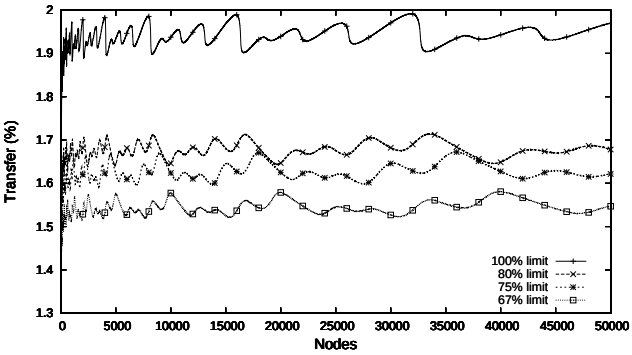

Figure 5. Transfer for $N = 4$ at various utilization thresholds.

Oversubscription with Threshold The bandwidth penalty can be reduced by adding pnodes when some target utilization is reached. Figure 5 shows the transfer rate for $N = 4$ and various utilization targets. The graph shows only the transfer rate since the guaranteed utilization coincides with the threshold except in the "no threshold" case which has a minimum of 85% (see Figure 4).

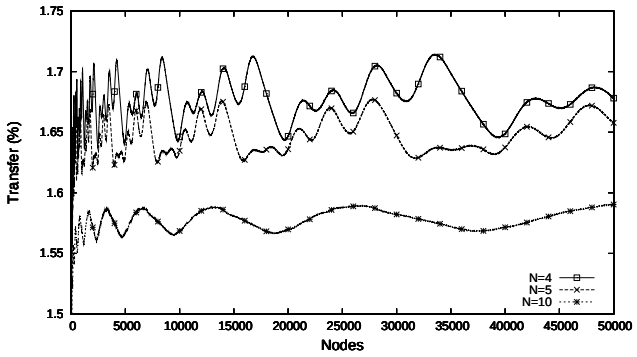

Figure 6. Transfer for various N when utilization is limited to 80%.

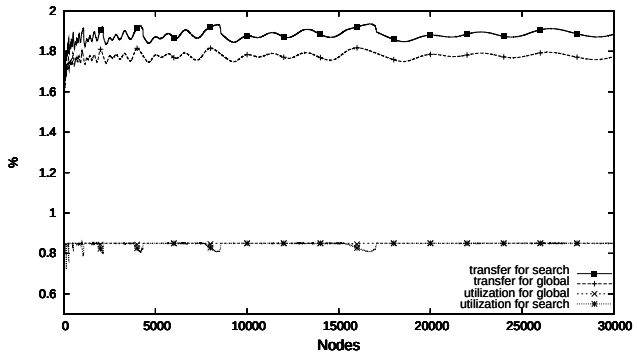

Figure 7. Impact of search for a transfer target, $N = 4$, 80% utilization threshold.

Impact of Number of Slots Figure 6 shows the transfer rate when we vary the number of slots for the same guaranteed utilization (80%). The results show that higher N generate lower traffic for the same utilization.

The experiments suggest that we should first decide the desired guaranteed system utilization and then derive the minimum value of N from the formula $U = (N-1)/N$. At the cost of more state per node, N can be further increased to reduce bandwidth requirements.

Impact of Non-global Search for Transfer Figure 7 compares a theoretical system in which the transfer target is located with the perfect global knowledge against an actual system with the transfer set limited to 100 elements on each pnode. Compared to the global case, the local scheme has a somewhat higher (6%) transfer rate, and occasionally lower utilization, showing that local search produces acceptable results.

Impact of Node Capacity Figure 8 shows the impact of node capacity on the transfer rate. Using $N = 5$ and an 80% threshold, we have varied the capacity of the nodes up to 1 million keys per pnode. With growing node size, the transfer rate increases slightly; this effect is negligible when pnode capacity increases from 32,000 to 1,000,000 keys.

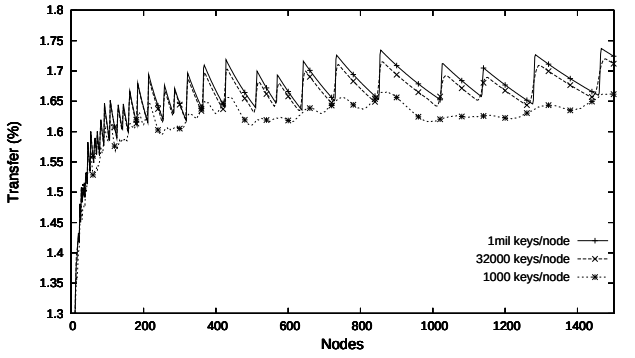

Figure 8. Impact of node capacity, $N = 5$ at 80% utilization.

6. Related Work

6.1. Network Construction and Routing

FPN is the first DHT that can effectively maintain fixed routing lengths while the system grows. Kelips [3] routes messages in a fixed number of steps, but only for a static system size. In Kelips nodes are divided into k affinity groups, and when k is $O(\sqrt{n})$ the path length is $O(1)$. As a result, Kelips can deliver fixed path length for each target system size, but unlike FPN, it cannot maintain fixed path lengths while the system grows. Moreover, the design is not intended for large size, since its memory consumption is $O(\sqrt{n})$ with an extremely large constant (approximately equal to the number of files stored on one node). This is because each node in an affinity group keeps pointers to all files stored on all other nodes in the group.

FPN is related to Internet address classification CIDR [10] in dividing a space spanned over bit vector into disjoint subspaces represented by variable-length prefixes.

Plaxton et. al [8] proposed an object location algorithm based on prefix routing. This scheme does not handle node failures, additions and deletions. Pastry [11] and Tapestry [4, 15] extend Plaxton proposal

by addressing these shortcomings. FPN bears resemblance to these systems as the key is divided in digits and the routing is done in general by resolving the next digit. However, in Tapestry and Pastry neighbors are defined for all values of a given digit provided they exist, whereas FPN maintains fewer neighbors since they are defined by bit position, not multiple digit values. As a result, neighborhood information is easier to collect and keep up-to-date in FPN. Routing tables for Tapestry and Pastry are similar to FPN's jump tables. However, jump tables are useful but not required for routing, whereas the routing tables in these two DHTs are necessary. Last but not least, digit size in the FPN is variable which allows for maintaining fixed number of hops as the network grows in size. In both Tapestry and Pastry routing path length grows logarithmically with the number of nodes.

In CAN [9] nodes are assigned zones in a d-dimensional torus where the number of dimensions d is constant for the entire system. The routing takes $O(dn^{1/d})$ steps and each node keeps $O(d)$ links. FPN has symmetrical complexities in relation to CAN - an FPN network with jump tables has $O(dn^{1/d})$ links and the routing takes d steps where d is the number of digits. eCAN [14] extends CAN to achieve $O(log\ n)$ routing independent of d.

Chord [13] and Kademlia [7] are DHTs in which routing takes $O(log\ n)$ steps and each node maintains $O(log\ n)$ links.

Finally there are DHTs with constant number of links, and routing messages in $O(log\ n)$ steps. Viceroy [6] topology combines approximate butterfly network and rings of predecessor and successor links. The routing takes $O(log\ n)$ steps with the number of links equal to a small constant (7 in this case). Koorde [5] is a variant of Chord maintaining only two neighbors per node and routing in $O(log\ n)$ steps.

6.2. Load Balancing

A basic load balancing technique in P2P is to keep multiple zones (virtual nodes) per one pnode. Chord proposes to keep $O(\log\ n)$ zones per node. This scheme ensures that with high probability the number of objects per node is within constant factor from optimal. However, this still can lead to poor minimum utilization. Rao et. al [8] propose several load balancing algorithms for P2P systems in which highly loaded pnodes transfer keys to lightly loaded ones until load balances. They can achieve high utilization, more than 90% with less than 50% data transferred. Unfortunately, they considered static network only, so it is not clear how often one will have to balance the load while the system grows. As a consequence, the total cost of continuous load balancing in terms of bandwidth consumption cannot be inferred. Moreover, they do not investigate the utilization during lifetime of the system.

PAST [12] is a file storage on top of Pastry. The achieved utilization rates are quite good, in excess of 90%, however at the price of some file insertion failures. The results reported there are not directly comparable to this work for several reasons: the experiment in the PAST case has been measured for a fixed configuration of 2250 nodes without looking at utilization while the system size grows; the entities stored are variable-size files versus fixed-size keys used for FPN experiments; and in FPN case no insertion failure is tolerated.

Another load balancing scheme is presented in [1], in which an object is mapped to a small constant number of peers rather than to a single peer, by using different hash functions. The insertion is done on the peer with the lowest utilization, and redirection pointers to it are inserted on all the others. Because the redirection pointers need update when the network changes, the scheme assumes that keys are re-published periodically.

7. Conclusions and Future Work

We introduced a new DHT called Fixed Prefix Network (FPN) designed specifically for future commercial applications built with the P2P paradigm. Exploiting the assumption that the resources used in this network are dedicated, we build FPN nodes with rich connectivity, resulting in fixed path lengths for extremely large range of network sizes. The short routing path is maintained even in the face of large fraction of failed nodes.

We introduced also a new storage load balancing scheme called oversubscription used to ensure high guaranteed storage utilization on the order of 80% for entire system lifetime by following simple rules locally about when new storage should be added This scheme has only a small extra cost in bandwidth consumption.

Unlike most of the related work, we have evaluated our system throughout the entire system lifetime instead of focusing on one specific system size. Our design is able to maintain fixed routing paths and deliver guaranteed utilization while the system grows.

Short routing paths are critical for user acceptance in commercial world where a good level of performance is expected. Ensuring high guaranteed storage utilization while limiting bandwidth consumption is absolutely necessary for commercial viability of the result-

ing systems if those resources are not free, as is the case for intended applications.

In the future we plan to extend the FPN to handle real world problems like network partitions and use the extended design as a platform for a prototype of one of possible commercial applications.

References

[1] J. Byers, J. Considine, and M. Mitzenmaker. Simple load balancing for distributed hash tables. In *2nd International Workshop on Peer-to-Peer Systems*, 2002.

[2] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a peer-to-peer lookup service. In *1st International Workshop on Peer-to-Peer Systems*, 2002.

[3] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[4] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proc. of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, Aug. 2002.

[5] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[6] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[7] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[8] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured P2P systems. In *2nd International Workshop on Peer-to-Peer Systems*, 2003.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of ACM SIGCOMM*, 2001.

[10] Y. Rekhter and T. Li. An architecture for IP address allocation with CIDR. RFC 1518, http://www.isi.edu/in-notes/rfc1518.txt.

[11] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329+, 2001.

[12] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[13] I. Stoica, R. Morris, D.Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the 2001 ACM SIGCOMM*, pages 149–160, 2001.

[14] Z. Xu and Z. Zhang. Building expressways for P2P. Technical Report HPL-2002-41, Hewlett-Packard Labs, 2001.

[15] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.