The Power of DHT as a Logical Space

Zheng Zhang
Microsoft Research Asia
zzhang@microsoft.com

Abstract—

P2P DHT has fast become a rather "classical" research field. The currently popular mindsets are mostly routing-centric, or storage-centric. In this paper, we argue that it maybe more interesting to simply view DHT as a logical space that can dynamically size itself with potentially unlimited amount resources. This is in someway analogous to the virtual memory in any contemporary operating system. We believe that exploring such a perspective will bring about new insights as well as applications. We illustrate the power of this abstraction with a few examples, including self-scaling, self-healing fat tree, self-tune storage system, and a light-weight quorum-based distributed lock protocol which does not assume constant number of members to start with. While their individual utilizations differ radically, all of them are united under this viewpoint.

I. Introduction

P2P DHT (distributed hash table) [14][12][9][14] has been one of the major focuses of various research communities today. There are several different perspectives that one may take upon P2P DHT. For instance, the network community views it as a promising overlay technology, and hence focuses on the robustness and efficiency of routing [10][8], and further builds value-adds applications such as application-level multicasting [22][1] and other interesting rendezvous services such as i3 [15]. System community, on the other hand, takes DHT as an abstract of state repository but with a definite spin on wide-area distributed environment, and thus has initiated a number of distributed stores such as Oceanstore[7], PAST[4], CFS [3]. Meanwhile, there are other interesting P2P applications in the web scenario, such as searching [11], spam fighting [21], even troubleshooting [16].

An alternative perspective that we propose in this paper is to view DHT as a single logical space that can be populated dynamically with unlimited amount of heterogeneous resources. Furthermore, it is possible to adjust the placement of both resources and other entities (such as objects or even software components) by appropriately manipulating their positions/addresses in the space. A possibly imperfect analogy is the virtual memory space of any contemporary OS: if one takes a snapshot, data/programs can live in different portions of the space that maybe substantiated by different levels of hardware hierarchy. We will offer a more extensive discussion in Section-II.

By emphasizing on this angle it is possible to both build systems and solve problems with a methodology that is superior to existing approaches. In this paper we illustrate our point by summarizing over a few projects we have been undertaking:

- A robust self-scaling, self-healing and self-optimizing fat-tree that organizes various resources into a hierarchy (Section-III).
- A self-tuned long-running object storage that explores heterogeneity of both resource power and object popularity for optimal performance (Section-IV), and can continue to evolve forward automatically. This is an example of self-configured and self-managed single-image distributed system.
- A light-weight quorum-based distributed lock protocol that can easily handle dynamic membership change (Section-V). Contrast to existing distributed consensus protocols in which dynamic membership changes are handled with a set of extra mechanisms, this protocol does *not* assume that the total number of quorum member is constant to start with.

The rest of the paper will go through the DHT property as viewed from the logical space point of view, and will discuss these three exemplary applications in detail. We will discuss related work in Section-VI and conclude in Section-VII.

II. DHT PROPERTIES AS A LOGICAL SPACE

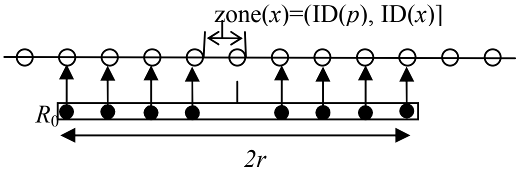

Figure 1. The simplest P2P DHT – a ring, the *zone* and the basic routing table that records *r* neighbors to each side.

A DHT, in its core, is a very large *logical space* (e.g 128bits) that can be partitioned dynamically with resources. Typically, a node joins DHT using a random ID (e.g. MD5 over its IP address). The ID, while random, remains fixed throughout. However, this is not a necessary condition. An ordered set of nodes, in turn, allows a node's responsible *zone* to be strictly defined. Let p and q be a node x's predecessor and successor, respectively. One definition of a node's zone is simply the space between the ID of its immediate predecessor ID (non-inclusive) and its own ID. In other words: *zone*(x) ≡ (ID(p), ID(x)]. This is essentially how consistent hashing assigns zones to DHT nodes [14] (Figure 1). This base ring (also called as *leaf-set*, as in Pastry[12]) is the simplest P2P DHT. To harden the ring against system dynamism, each node records r logical neighbors to each side. These states form the basic routing table, and are updated to keep the invariant when node join/leave events occur.

The lookup performance is O(N) in this simple ring structure, where N is the number of nodes in the system. Elaborate algorithms built upon the above concept achieves O(log$N$) performance with either O(log$N$) or even constant states (i.e. the routing table entries). Representative systems including Chord[14], CAN[9], Pastry[12] and Tapestry[17]. As it turns out, for a number of DHT applications that we will discuss in this paper, lookup performance is not critical. The most important and relevant point is the integrity of the DHT space: it does not have hole except transient jitter when node join/leave (i.e. membership change).

Many distributed system requires consistent and global membership, and this is the chief reason that hinders their scalability. This is not the case for DHT: when membership change occurs, only the leafsets affected need to be updated. This effectively constraints system churns in a local scale. In other words, DHT enjoys unlimited scalability precisely because overlapping and localized leafsets. O(log$N$) routing table and performance are critical *performance* optimizations, which may not be necessary, or sometimes may not be enough. For instance, XRing[20] is a P2P DHT we have built that gives O(1) lookup performance with O(N) state and is tailed for environment where either churn is low or system scale is small.

Another popular notion is that node/object ID/keys are purely random and stay constant once they are set. This needs not be the case, either. Consider that we want to organize a DHT-based storage with 5 nodes, the first four have capacity one-fourth of the fifth node. Assuming object keys are random. We would like the space to be partitioned such that the first four nodes occupy the first half of the space, each taking 1/8-th of the total space, while the last node takes the second half. Random node ID will leave small capacity nodes oversubscribed while having the large capacity nodes underutilized. Likewise, the other option is to keep node ID random, but skew the object key distribution. To this date, there has not been an entirely satisfactory solution to this problem. This is not the problem that this paper attempts to address. In the example below, however, we will show the freedom of manipulating node and object ID/keys can help to construct more flexible systems.

III. SELF-SCALING HIERARCHY

Hierarchy is one of the most fundamental topologies to organize a set of nodes. Typically, the higher level nodes are more capable/powerful and consequently assume more controls and responsibilities. From this perspective, a client-server architecture is a two-level hierarchy with a very fat base. Conventional tree-building process operates in the discrete, physical entity domain and faces many classical challenges such as leader election and tree-stabilization.

A DHT-based self-scaling hierarchy approaches this problem rather differently, and is more flexible and robust. The idea is to "draw" the desirable topology in the logical space first, and the *logical tree nodes* then map back to the machines that comprise the DHT. Once the mapping is done, building the tree is straightforward by finding child-parent pairs. As we will illustrate shortly, all operations are done in a completely distributed fashion. What is required is a very rudimentary DHT: a ring suffices just as well. Each participating node initially selects an ID in the range of [0..1] and divides the space uniformly in the manner of consistent hashing as in Chord[14]. This is one example where O(*N*) lookup performance does *not* matter.

We proceed to describe how to build a *self-scaling, self-organizing fat-tree.* A fat-tree is one where there are more nodes in the higher levels of the tree. To be self-organizing, the tree should be built automatically. Likewise, to be self-scaling, if *N* increases then more levels should appear and the higher level should grow more nodes. This would be quite a difficult task to accomplish with traditional approaches.

To prepare the fat-tree topology, the logical space is recursively divided by a factor of *k*. The level-1 partition is the entire space, and the level-2 partitions are *k* partitions each of 1/*k*-th of the total space, and so on so forth. Assuming a partition has the boundary [*a*, *b*], and the center of the partition is at $p=(a+b)/2$, then the logical tree nodes are the points in the segment $[p-\Delta/2, p+\Delta/2]$, where $\Delta = c\cdot(b-a)$, and *c* is a real number between [0, 1] (e.g. 1/16). These segments are shown as thick bars at Figure 2 . "Drawing" this logical tree is a local operation that each machine in the DHT can perform at any point of time, as long as they know *k*, *c* and Δ.

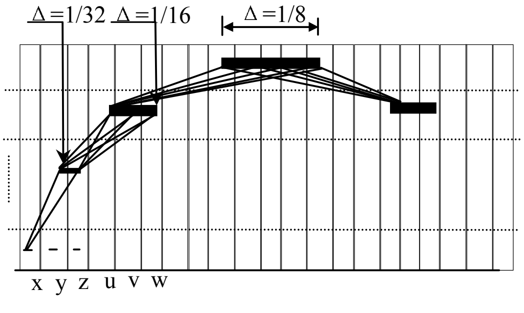

Figure 2: self-scaling fat tree. For simplicity we only draw part of the tree. Δ=(b-a)/8 where [b,a] are the boundary of a partition. The partitioning in levels is binary. Note that both *y* and *z* are interior nodes of the resulting hierarchy rather than leave nodes.

Next comes the procedure to *map* logical tree nodes back to the machines in DHT. Each machine intersect its zone with the logical tree (the bars) generated, and thus identify its position(s) in the tree. If it shall appear at multiple levels, then it picks the bar at the *highest* level; this will be its position in the final fat-tree topology. In Figure 2, for instance, node *z* and *w* will be at level 3 and 2, respectively. At this point, all machines have identified their positions in the topology.

The final step is to connect the parent and child nodes, and this is where the DHT lookup operation is needed. Each machine identifies the next higher level bar of the bar it is representing in the hierarchy. For node *z* in our example, this will be the bar at level-2 since its own bar is at level 3. The machine then takes the target bar as input, and uses DHT lookup to identify the machines that will serve as its parent in the hierarchy. It then establishes connections, which are maintained with heartbeats. In our example, both *y* and *z* will connect to their parents *u, v* and *w*.

To see that the tree is robust, consider the case that a node departs. We only need to worry about its role as a parent. DHT ensures that the departed machine's zone will be taken over by one of its logical neighbor machines. The departed node's now orphaned children, will simply redo the same procedure as before and find their new parents. As such, the hierarchy is self-healing.

The tree is self-scaling: more levels appear when new nodes join and partition the space further. The machine whose zone gets split with the new one simply breaks all current connections and thus force its current children to refresh their topology; it may need to find new parents too shall its level gets affected.

It is also easy to see that this construction accomplishes a fat tree. At higher level, the logical tree nodes are more dispersed and have more probability to be hosted on different physical machines. The larger the *c*, the fatter the hierarchy becomes. A special case is *c*=0, in which case at any given level there is only one tree node. This self-scaling, self-healing hierarchy is first introduced as SOMO (*self-optimized metadata overlay*) [19], but with a top-down building procedure.

The self-optimizing aspect includes two aspects. The first is to select nodes according to a given metrics. For example, nodes with higher power/bandwidth should go to upper level. Note the topology itself is a tree and thus merge sort can be applied. Once the election is done, the more powerful set of nodes can take their target positions by swapping their ID with those who currently occupy these positions (via rejoining the DHT). Note how the fact that we relax the ID selection allows the tree to self-optimize.

In MSR-Asia, we have organized hundreds of lab machines with a simple DHT, and then layer a hierarchical, in-system monitoring infrastructure over them. This infrastructure allows us to periodically gather various performance statistics from the machines in a soft-state manner, and then aggregate them up to the tree and finally stream to a database. The same hierarchy allows us to send scheduling and maintenance instructions down to the machines. As will be discussed in the next section, it is also used as a component technology in the self-tuned storage system that we are building.

IV. SINGLE-IMAGE, SELF-TUNED DISTRIBUTED SYSTEM

The flexibility of a resource and object to choose locations to join a DHT space gives the possibility of self-tuned, single-image system. We have designed and are developing a DHT-based long-running object storage called RepStore, made up by commodity PCs. While there are already many proposals for DHT-based storage, the most important differentiating point of RepStore is its self-tuning aspect.

The total cost of ownership of IT infrastructure is not hardware, but the management complexity and the overhead it brings. To take a more evolutionary perspective, the simple fact is that for any large, long-running system, machine profiles will change over time (obsolete ones retire and new ones procured and installed); and so does object popularity. In other words, heterogeneity exists in both resources as well as object profiles. Thus, our goal is to align object popularity and machine power across the *whole* system, and do it in a fully distributed manner. Furthermore, self-tuning should be accomplished continuously and online without interrupting normal operations. From end-user's point of view, they just drop the objects into the store and expect the system to return the right level of performance and reliability. Likewise, system administration should do no more than taking away dead/obsolete machines and plug in new ones, all the while expect the system to improve and upgrade itself online.

RepStore, for the time being, is targeted for enterprise internal environment with dedicated machines. As such performance does matter, and this is why O(log*N*) lookup performance is neither sufficient nor necessary (due to the low churn rate). Consequently, RepStore is layered on top of XRing, leveraging its O(1) lookup performance.

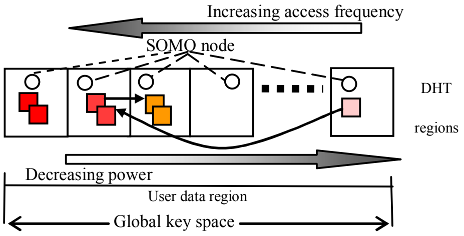

Figure 3: aligning object popularity with machine power

Region is RepStore's way of dealing with heterogeneity (Figure 3). The idea is that when the total key space of a DHT is large, or in case like CAN is practically infinite, we can partition the key space and group machines into regions, and then assign objects into the regions such that they are served by the most appropriate class of machines. We use N_r to denote number of regions, typical value is 4 or 8 – roughly corresponding to number of different generations of machines that will be kept on duty at any given point of time. The IDs of machines as well as object keys are composed by logN_r *region id* that is followed by a 160bit hash over its IP address/content. An object is stored on the node that hosts its key, and is further replicated to several logical neighbors of the node to guard against failures. Also, SOMO is deployed over the DHT to perform both monitoring services as well as to trigger optimizations (described shortly).

The high-level idea is very simple:

- The first we do is to classify machines into N_r classes and order their capacities/power. Machines of the same class join the same region. This is a simple and practical way of grouping machines of the similar profiles. When a class of machine retires, its region is re-filled with new generation of machines and this is how RepStore evolves with resource capacities. Notice that we can always find the profile of machines by querying SOMO, allowing an arbitrary machine to join the most appropriate region by setting its region-id bits.
- Objects, too, are deposited into the right region according to their popularity. They are initially

randomly placed into a region and later migrated to a region according to its rank in the popularity. The popularity is found by periodically gathering object access frequencies reported through SOMO. When an object is found to be misplaced in a different region (for instance it is among the top5% but is placed in a region whose machine power has rank 3), it is migrated to an appropriate region (in this case, the region with the most powerful machine). Access to an object is done by caching its ID at the client application, and if that fails due to object migration, we multicast the request to all the other N_r-1 locations that the object can possibly exist.

In this way, machine power and object popularity is gradually aligned. Furthermore, there is no disruption to normal operations and the system can continue to upgrade and improve the performance with minimum administrative attention, and we can also reasonably ensure objects with different popularity are served differently, yielding a crude level of quality of service (QoS) guarantee. The system, therefore, evolves mostly by itself.

Notice that RepStore accomplishes its goal not by magic, but by the very facts that 1) DHT is a single and huge addressing space and 2) both resources and objects can be freely relocated by setting their ID/keys.

V. Simple quorum-based Distributed Lock Protocol with Dynamic Membership

The previous two sections describe the utility of DHT in a local environment for distributed system. In this section we turn our attention to an entirely different problem: the classic algorithmic problem of distributed consensus. In particular, we are interested in building a distributed lock with quorum consensus protocol.

Distributed quorum is a fundamental building block of many wide-area applications. Conventional approaches take majority consensus approach and they work, essentially, by passing an agreement from m out of n entities. While the theories are well understood [6][2][5], the long standing problem is the one of flexibility: what if we need to size n dynamically? More seriously, what if the number of faulty nodes exceeds the threshold m and, as a result, makes the system hang altogether? The state-of-art way of dealing with membership change has been to relying agreement among *all* existing (non-faulty) members to institute new membership change. This is both costly and complex and, as a result, not very scalable. We would like to have a much light-weight approach.

As a fundamental departure from previous approaches, we believe that the key to solve the dynamic membership problem is to investigate a mechanism where constant number of members is not required to start with.

The basic idea turns out to be quite simple. All the above issues would be largely taken care of if the quorum members (or nodes – the two are used interchangeably for the remaining of this section) are organized into a DHT and we require more than *f=m/n fraction* of the logical space agree. This casts the quorum system to be run in a continuous space rather than a discrete one, and instantly brought many benefits. However, before we can make the claim we must specify more clearly our goal and assumptions, which are not any different than those of the existing protocols.

Goal: The consensus is purely defined from the client perspective, and our goal is that, with high probability (defined later), at any given point in time, there is only one consensus in the system. We want to use the consensus to implement a distributed lock protocol.

Assumption: we assume non-malicious client. That is to say, it does not propose different values to different entities in the DHT. The whole logical space has the bound [0..1]. Messages can get lost, take arbitrary time to deliver, get duplicated but they are not corrupted.

The protocol works as follows. The nodes in the DHT execute a simple state machine. A new node will have its value initialized to null. A node with the null state can accept a proposal by setting its value to be the one enclosed in the proposal. The node then responds with an “accepted” message which includes the node’s *share* of space, namely its zone size, and a lease. A node that has already accepted a proposal can not accept new proposal, but can reset its value to null if the client who won its acceptance 1) sends a “release” message and 2) does not renew its lease.

The client drops a lock request into the DHT as a proposal whose value is the client’s ID. The request is broadcasted to all the DHT members in a best-effort fashion. The broadcast can use any

DHT-based multicast trees [22][1]. The client then collects the responds and tallies the shares. If, before a pre-defined timeout period, the total shares with the value that it proposed exceeds f (f>1/2), then it considers itself the winner of the lock. Otherwise it sends release messages to all nodes that accepted its proposal, backs off randomly with a delay, and retries again. The delay is necessary to resolve any deadlocks, and the duration is set to be inversely proportional to the total shares that do agree its proposal.

This is clearly a probabilistic approach since many things can go wrong during the process. For example, the request may be lost (or not received before the timeout), the node could vote for proposals from other clients, causing multiple, possibly contradicting consensuses. Similar problems arises when membership changes: a node may quit and the takeover node now commands a larger share than the vote it has issued; a node may join and therefore the node it splits against loses a portion of share without notifying the previous client.

All these faulty cases – as well as the one where the nodes act maliciously can be studied with probability theory without hairsplitting the detailed causes. The question in concern is the odds that there exist in the system multiple winning consensuses of different clients. The condition that a client A believes it has won means it has collected f fraction of the space. The remaining 1-f can be collected by another client B. If, among the f fraction of the space, defects occur such that x fraction further gets "stolen" by B, then if x+1-f now belongs to B, B may get another winning consensus. Therefore, to ensure we can tolerate x fraction of defects, we must have $x+1-f<f$, and thus $f>(x+1)/2$. As a matter of fact, its discrete version works out exactly as many Byzantine protocols guarantee, i.e., to tolerate m failures, one must have $3m+1$ nodes.

This protocol is more resilient to attacks than those relying on fixed number of members, which will be faulty if more than m members are compromised. In this protocol, it is possible to continuously drop in new nodes so as to increase the portion of good nodes and hence the robustness of the system. This now becomes a matter of speed: instead of having a faulty protocol, as long as we can institute good nodes sufficiently fast, we will have a correct protocol. However, this will only work if and only if all that node's share (its zone size) is verifiable and authentic, which is a big enough assumption all by itself.

In addition to solving the dynamic membership change problem, nodes with higher trust level can take more shares by commanding larger zones and thus assume higher responsibilities in the agreement process. In other words, weighted quorum consensus can be easily supported.

VI. RELATED WORK

From a logical space point of view, the most important integrity guarantee is the leafset, which is present in virtually all DHT proposals DHT [14][12][9][14] but this paper argues that leafset should be given a more central role. A light-weight leafset protocol that maintains the integrity of the DHT space is proposed in [18].

Various O(log$N$) routing proposals, in essence, build routing entries by drawing a distributed binary search trees in the logical space. To our knowledge, a self-scaling, self-healing fat tree built with DHT has not been proposed before. Also, none of the exiting DHT-based storage systems [7][4][3] has the self-tuning aspect yet. Up-to-date, our sigma protocol [13] is the only distributed lock over P2P that we know of; the work presented in this paper differs from Sigma which assumes a fixed set of logical virtual servers spread in the DHT space, but the basic idea and system model is similar.

VII. CONCLUSION AND FUTURE WORK

The network community, in general, views P2P DHT as an overlay; the system community takes the perspective with a more storage-centric mindset. This paper argues that it would be interesting to examine the potential of P2P DHT as a single-address, large virtual space, analogous to virtual memory in an operating system. We have explained why this is so by describing a few systems and protocols that would be difficult to achieve otherwise. We believe that, especially in the realm of distributed system and algorithms, we will continue to find new utilities of P2P DHT by exploring this perspective.

Acknowledgement

This paper draws inspirations and ideas from many staff members and students that the author has worked with. Within MSR-Asia, Yu Chen proposed the solution to optimize SOMO to get rid

of redundant tree node, Shiding Lin and Chao Jin have worked on RepStore. Students from Tsinghua University such as Shuming Shi, Qian Lian and Ming Chen have all contributed to these projects. The author also benefited from many enlightening discussions with colleagues from other MSR colleagues including Lidong Zhou, Wei Chen, Butler Lampson and Leslie Lamport.

References

[1] Castro M., Druschel P., Kermarrec A., and Rowstron A. SCRIBE: *A Large-scale and Decentralized Application-level Multicast Infrastructure.* IEEE Journal on Selected Areas in Communications, Vol. 20. No 8. Oct. 2002

[2] M. Castro, B. Liskov, *Practical Byzantine Fault Tolerance.* in Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, February 1999.

[3] Dabek, F., et al. *Wide-area cooperative storage with CFS.* in Symposium on Operating Systems Principles (SOSP). 2001. Banff, Canada.

[4] Druschel, P. and A. Rowstron. *PAST: a large-scale, persistent peer-to-peer storage utility.* in HotOS-VIII.

[5] E.G.Kotsakis and B.H.Pardoe *Dynamic Quorum Adjustment: A Consistency Scheme for Replicated Objects* Proceedings of the Third Communication Networks Symposium, July 1996.

[6] L. Lamport, R. Shostak and M. Pease, *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, 4(3):382-401, July 1982

[7] Kubiatowicz, J., et al. *OceanStore: An Architecture for Global-Scale Persistent Storage.* in ASPLOS 2000. 2000. MA, USA: ACM.

[8] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, I. Stoioca, *The Impact of DHT Routing Geometry on Resilience and Proximity*, in ACM SIGCOMM'03.

[9] Ratnasamy, S., et al. *A Scalable Content-Addressable Network.* In ACM SIGCOMM. 2001. San Diego, CA, USA.

[10] Ratnasamy S., Shenker S. and Stoica I. *Routing Algorithms for DHTs: Some Open Questions.* Proceedings of IPTPS 2002

[11] Reynolds, P. and Vahdat, A.. *Efficient Peer-to-Peer Keyword Searching.* Middleware, 2003.

[12] ROWSTRON, A., AND DRUSCHEL, P. *Pastry: Scalable, distributed object location and routing for large scale peer to peer systems.* Proceedings of IFIP/ACM Middleware (Nov. 2001).

[13] S. Lin, et al, *A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems*, in IPTPS'04.

[14] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. *Chord: A scalable peer to peer lookup service for internet applications.* In Proc. ACM SIGCOMM (San Diego, 2001).

[15] Stoica, I., et al. *Internet Indirection Infrastructure.* in ACM SIGCOMM. 2002.

[16] Wang J. H. et al. *Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting.* In IPTPS'04

[17] Zhao, B., Kubiatowicz, J.D., and Josep, A.D. *Tapestry: An infrastructure for fault-tolerant wide-area location and routing.* Tech. Rep. UCB/CSD-01-1141, UC Berkeley, EECS, 2001.

[18] Zhang, Z. et al. *Leafset Protocol in Structured P2P Systems and its Application in Peer Selection.* Microsoft Technical Report.

[19] Z. Zhang, S. Shi, and J. Zhu, *SOMO: Self-organized metadata overlay for resource management in P2P DHT*, 2nd International Workshop on Peer-to-Peer Systems, Berkeley, CA, USA, Feb. 2003.

[20] Zhang, Z. et al. *XRing: Achieving High Performance Routing Adapatively in Structured P2P.* Microsoft Technical Report.

[21] Zhou, Feng et al. *Approximate Object Location and Spam Filtering on Peer-to-Peer Systems.* Middleware 2003.

[22] Zhuang S.Q., Zhao B.Y., and Joseph A.D. *Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination*, NOSSDAV'01, New York, USA