The Quest for the Holy Grail of Aggregation Trees: Exploring *Prefix Overlay(d)* Treasure-Maps

Vitus Lorenz-Meyer and Eric Freudenthal

Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
emails {vdlorenz, efreudenthal}@utep.edu

Abstract

Collecting data from a network of distributed information sources is common to a multitude of problems. A common problem in these systems is the sheer amount of data that can easily overload any single node. A possible solution to this problem is to reduce the data by aggregation close to its source. As this is most easily done through an overlayed reduction tree most systems build an overlay for this.
With the widespread use of Peer-to-Peer (P2P)-style applications this has become evermore complicated, because of their highly dynamic nature. Distributed Hash Tables (DHTs), also called *Structured Overlays* (for example CAN [6]), have solved the problem of routing in dynamic, unstructured networks.
In this technical report we present a solution to the problem of maintaining an efficient reduction tree in a dynamic network using a structured overlay.

Distributed systems are generally designed to satisfy a target problem's requirements. These requirements frequently do not include instrumentation of the system itself. As a result, the designs typically do not include instrumentation infrastructure, which makes tuning and debugging difficult. Furthermore, focusing upon adding this infrastructure might distract an engineer from designing for performance and thus can introduce an unacceptably high overhead to the system even when it is not used - compared to a system designed without instrumentation in mind. The objective of PlanetenWachHundNetz (PWHN) is to develop infrastructure suitable to assist the instrumentation of self-organizing distributed applications that lack data collection and aggregation mechanisms.

The instrumentation and tuning of P2P systems can require the analyses of data collected from a large number of nodes. If the data has not been selected and aggregated properly by the providing nodes, the resulting amount of data may be too large to use available communication and storage capabilities. Moreover, if it is sent to a single, central collection point for analysis, that node's resources might become saturated. Therefore, centralized solutions may be impractical. Thus, it may be desirable to aggregate summaries of data in a distributed manner, avoiding the saturation of a "master" host who would directly transmit instructions to and receive data from all of the other (slave) nodes.

Google's MapReduce was designed for the distributed collection and analysis of very large data-sets, in environments with groupings of participants located in dedicated computation centers that are known a-priori. A user of MapReduce specifies arbitrary programs for its map and reduce phases. The *Map* phase "selects" the data to be aggregated and outputs them in the form of intermediate key/value pairs; afterwards, the *Reduce* phase digests these tuples grouped by their key into the final output. These selection and aggregation programs can frequently be easily constructed using the many convenient scripting programs commonly available on unix systems and then linked against the MapReduce library (written in C++).

PWHN extends the MapReduce model to P2P. The primary difference between the environment MapReduce was designed for and P2P-networks is that the set of active participants and their logical groupings frequently changes in P2P systems. Conventional approaches of pre-structuring aggregation and distribution trees are inappropriate given P2P system's large *churn*[1] of membership.

[1]Churn means the characteristic of a normal P2P system that nodes frequently join and leave; thus, the membership "churns"

A measuring infrastructure for distributed systems faces the challenges of selection and aggregation of relevant data under churn, while ensuring good usability.

PWHN utilizes P2P techniques including Key-Based Routing layers (KBRs) and a data-structure that extends existing algorithms for guiding the construction of an aggregation tree upon the internal structure of DHTs.

0.1 Coral

Consider Coral [1], a load-balancing P2P-Content Distribution Network (CDN) implemented as a HTTP proxy for small-scale servers that cannot afford large-scale dedicated solutions like akamai. It is referenced by many private websites and provides high availability. Like many distributed applications Coral's construction does not include instrumenting and logging infrastructure, thus it is hard to hard to tune and determine its limits of scalability. After this was discovered, a logging and collection infrastructure was crafted onto Coral. This approach was centralized, a central server outside of Coral just requested all logs and inserted them into a database. This technique did not scale well and was therefore discontinued quickly.

We learned of Coral's misfortune and decided that it is a hard problem to collect and aggregate statistics about a P2P application. It was my motivation to build a system to make this process easier for application designers.

1 The world before Prefix-Suffix

This section introduces two different data-structures that researchers use to build an aggregating system on top of a key-based structured overlay.

1.1 Quester 1: KBT

Almost all DHTs, sometimes referred to as *Prefix-based Overlays* or *Structured Overlays*, build on Plaxton et al.'s [5] groundbreaking paper (because of the first letters of the author's surnames - Plaxton, Rajaraman, and Richa - known as *PRR*) on routing in unstructured networks.

Key-Based Trees (KBTs) use the internal structure of the trees that Plaxton-style [5] systems automatically build for routing in the following way. These key-based routing protocols populate a flat identifier space by assigning long bit-strings to hosts and content which are simply 160-bit integer identifiers. Two keys that have n most significant bits of their IDs in common are described to have a "common prefix" of length n. Thus, "prefixes" of IDs function as "search guides" since they are prefixes of actual node keys. Now consider the following. Each node is the root of its own "virtual" tree. Both nodes at depth n have $(n-1)$-bit prefixes in common with their root, while the next (n^{th}) bit is 0 for the left and 1 for the right child. The result is a global, binary tree.

This assumes that the DHT fixes exactly one bit per hop. DHTs that fix more than one bit per hop will have a correspondingly higher branching factor. Since actual nodes will always have complete bit-strings, all internal nodes that are addressable by a prefix are "virtual," in the sense of the tree. The physical nodes are the leafs of the tree and can be reached from the root ("empty prefix") by a unique path. Since the relationship between tree nodes and their place in the ID-space is unambiguous, meaning that the tree is fully defined by a set of node keys, I have termed this data-structure Key-Based Tree (KBT).

Each DHT defines a distance metric that defines "nearby" nodes and guides searches.

Two examples of systems that use KBTs are Willow [7] and SOMO [9].

1.1.1 Classification

The designers of an implementation of a KBT define its tie-breaking behavior, meaning which child will be responsible for its "virtual" parent, since both children match their parents' prefix. There are two basic approaches.

1. *Static* algorithms are deterministic for the same set of nodes and can only change if that set changes.

2. *Dynamic* algorithms determine the parent based on characteristics of the involved nodes, i.e. "in-vivo" (may even be evaluated in the form of queries). This has the effect that the choice is not deterministic, i.e. can be different for the same set of nodes, and can change during its lifetime.

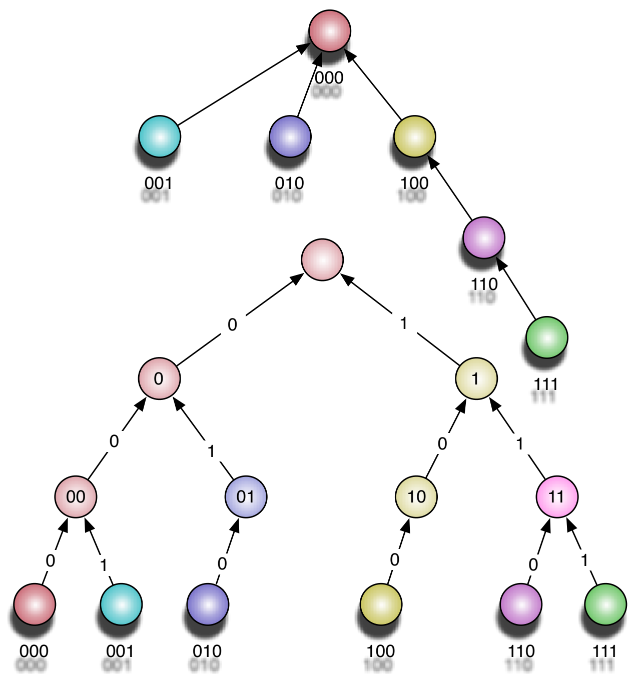

Figure 1: Sample KBT with six nodes using a 3-bit wide ID-space and a static approach in which the left child assumes the parent role ("Left-Tree"). The bottom part of the figure shows the "virtual" tree, whereas the top part shows the resulting real messaging routes.

Choices of a dynamic approach are based on age, reliability, load, bandwidth or based on a changeable query. Static algorithms can only choose the node that is closest to a a deterministic point of the region (e.g. left, right, center). A static choice has the advantage of predictability, provided you have a rough estimate about the number of nodes. This predictability allows making *informed* guesses about the location of internal nodes in the ID space.

1.2 Quester 2: FTT

Routing towards a specific ID in key-based, structured overlays works by increasing the common prefix between the current hop and the target key until the node with the longest matching prefix is reached. The characteristic of

DHTs, that a message from every node towards a specific ID takes a slightly different path, leads to the realization that the union of all these paths represents another tree which covers all live nodes - a different one for each ID. Thus, unlike KBTs, where the mapping nodes and the tree is fully defined by the set of node IDs, this tree is ambiguous - i.e. is dependent on finger table content.

These trees are like inverted KBTs; the exact node with the longest matching prefix is the root, whereas in a KBT every node could be the root depending on its tie-breaking algorithm, since every ID matches the empty prefix. Since this tree depends on the fingertables of all live nodes, I call it a FingerTable-based Tree (FTT).

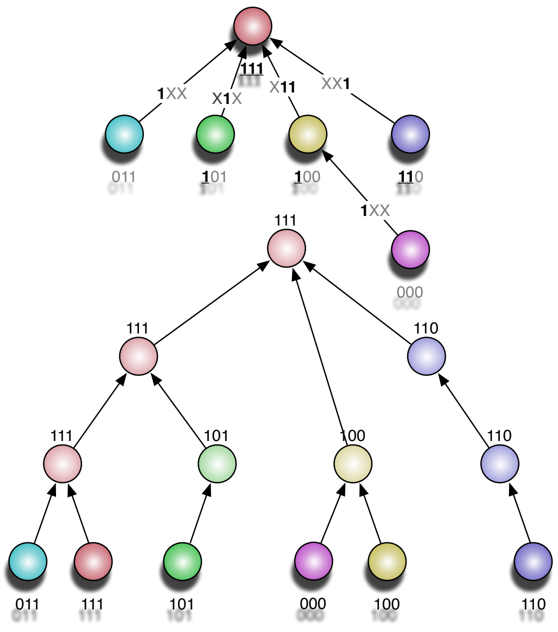

Figure 2: Top: Example graphical representation of all paths from 6 nodes towards Key(111) for a 3-bit wide ID-Space (the matching prefix is highlighted on each node and the corrected bit on each route), and the resulting FTT (bottom)

Example systems that use FTTs are PIER [2, 3, 4] and SDIMS [8].

1.2.1 Classification

FTTs can be classified by their result-forwarding strategy: i.e. at what time aggregated results are sent to the parent. There are two possible options for this parameter:

1. *Directly after child-update*: The new result is sent to the parent directly after a child sends new data. This necessitates caching of incomplete results for running queries.

2. *After children complete*: New result is only sent to the parent after it is sufficiently complete (or stabilized). This necessitates a guess about the number of children which is another parameter when using this option.

KBTs are binary if the underlying DHT fixes one bit per hop and every node will be at the same depth i, with $i = Number\ of\ bits\ in\ ID$. But generally they are not balanced.
FTTs on the other hand will never be balanced, and not all nodes will be at the same level. Moreover, they do not have to be binary. However, FTTs have the advantages of allowing more than one FTT at a given time, and the ability to being rooted at any node over KBTs.
These properties make the usefulness of both trees as aggregation aids questionable.

2 The Quest for Aggregation: Engineering the Holy Grail

2.1 Competitor's skeletons lining the mountain way

The Problem with both the aforementioned types of trees is that they do not have to be regular. While a FTT is certainly the better data-structure for aggregation, it even does not have to be binary. But being binary or having a strong upper bound on the branching factor is a characteristic which comes in very handy in trees that are meant for reducing data, because it helps preventing 'hot-spots' in the tree. *Hot-spots* are introduced by uneven load distribution in

the tree, i.e. some nodes that have higher fan-in than others.
A FTT that is not binary results from the underlying DHTs flexibility in chosing the next hop while routing. This flexibility ensures that a DHT is able to maintain its routing invariant. But at the same time it severely restricts the applicability of a DHTs underlying tree-structure to aggregation. We would like to subtly change the DHTs tree or build a new one on top of the KBR in such a way that it is regular only for reducing purposes while still preserving the KBRs ability of efficient routing.

2.2 In the Hall of the Mountain King: KMR

We seek a data-structure that combines the FTTs load-balancing feature with the determisn of a KBT. Since the FTT is defined by the entries of all participating nodes' fingertables, it is not predictable and may fluctuate frequently. This stems from the fact that the DHT needs some flexibility to be able to keep routing in the presence of churn.

A DHTs flexibility in routing means that it can pick the next hop from all those that have one more matching prefix bit. Thus the length of the prefix is inversely proportional to the number of candidate nodes. Consequently, the length of the *suffix* (the remaining bits after the prefix) is directly proportional to the number of eligible nodes. For k suffix bits the number of candidates is 2^k.

This leads to a solution which consists of a KBT rooted at a particular key. Since the number of candidates is directly proportional to the length of the suffix and we would like this number to be exactly one, we need to reduce the length of the suffix. This can be done by "fixing" it to the current node's suffix.

For example, in a "normal" DHT, a node with ID$=000$ that wants to route a message to Key(111) has four choices for the next hop, those that match $1XX$. Fixing the prefix and suffix to those of the current nodes, while still flipping the next bit (i.e. adding one more prefix bit) essentially reduces the number of routing candidates to one. Continuing the example, node(000) will have to route the message to node(100) because that corrects the next bit of the (currently empty) prefix and keeps the same suffix. This ensures that the resulting tree will be balanced and have a strong upper bound on the branching-factor of $len(ID)$ for a fully populated ID-space. This data-structure bears strong similarity to MapReduce, albeit keyed with a certain ID, thus I call it Key-based MapReduce (KMR).

A KMR is a subset of a KBT because a KMR is fully described by the set of nodes and the root key, whereas the KBT can have any permutation of the nodes with the constraint that all nodes in a subtree have the same bit at the respective level of the subtree's root. For n levels $a = 1 \cdot 2^n + 2 \cdot 2^{n-1} + \cdots + 2 \cdot 2^2 + 2 \cdot 2^1$ different KBTs are possible. Specifically, this means that a KBT with a set of nodes has a chance of $p = \frac{1}{a}$ to look like the KMR with the same set of nodes.

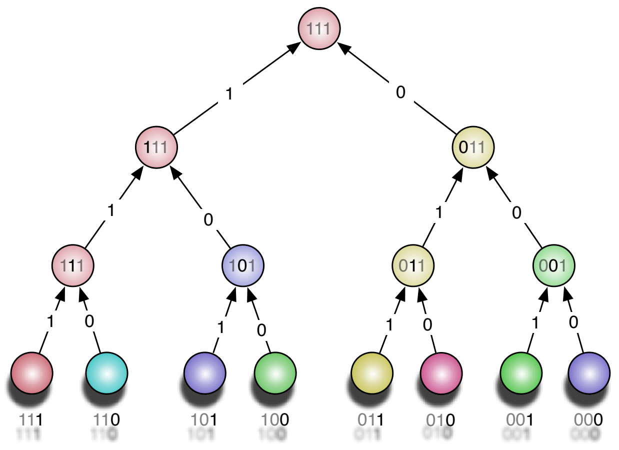

Figure 3: Example of a KMR (the flipped bit is highlighted at each level) for a 3-bit wide ID-Space, all nodes present, for Key(111)
(For homogeneity I will always represent KMRs such that the root node is the left child, thus the right child always has the corresponding bit of the root flipped. This structure is sometimes called a *Left-Tree*.)

2.2.1 The long way down

In the dissemination phase, a message has to be routed from the root of the tree to all leafs. It is done recursively. This is particularly easy due to the characteristic of a KMR that each internal node is its own parent if it has the

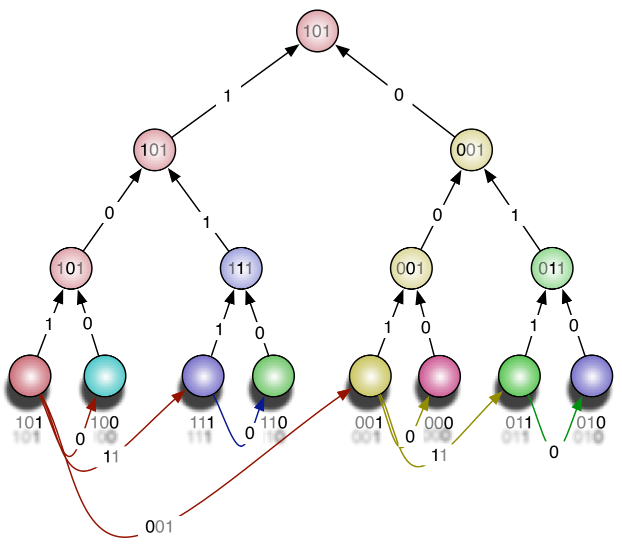

Figure 4: Explanation of the dissemination phase of a KMR rooted at Key(101)

same suffix as the root at the corresponding level, which in my representations will always be the left child. Each parent (root of a subtree) will be its own left child all the way down, and thus only needs to send the message to each of its (right) siblings. Since the length in bits of the suffix of a node will determine its level in the tree, this is the number of messages it needs to send. Thus, every node sends exactly P messages, where $P =$len(suffix), to other nodes according to the following algorithm.

Listing 1: Algorithm for the dissemination phase of a KMR

```
for(k = len(suffix); k >= 0; k--)
        route( OwnID ^ (1 << k) );
```

The algorithm starts at the node with the longest matching prefix and flip bits in the prefix “from the front,” thus doing the opposite from the aggregation phase: subtracting bits of the matching suffix. The algorithm sends as many messages as there are bits in the prefix.

Unfortunately, most DHTs will not have a fully populated ID-space. To prevent nodes from trying to route to a large number of non-existent nodes, which in a normal DHT would end up storing the same value over and over under different keys at a few nodes, an application building a KMR on a DHT would try to get access to the underlying fingertable and KBR Layer. There are two ways to solve this dilemma.

First, adding a single function to the DHTs interface allows an application to build a KMR on it. Since the application needs to find nodes without actually storing data on them, which would be the (unwanted) side-effect of a `put`, the function takes a key and returns the closest node to that key. Thus, it is called FIND_NODE.

After the parent of a subtree has determined the closest node to one of its children it was looking for, it just sends the message to that node instead (let us call that node A). This node has to assume the role of its non-existent parent and resends the message to all the children that its parent would have sent it to. Fortunately, due to the fact that the node, by virtue of getting this message for its parent, now knows that it is the closest ancestor to its parent, it is able to discern which of its siblings cannot be there. These are all the nodes who would have been closer to the original parent. Thus, A only has to resend the message to all of the direct children (of its parent) whose IDs are further from their parent's ID. In the representation I use in this paper, these happen to be all of A's parent's children to A's right.

The aforementioned leads to the second solution. If the application has access to the KBR and has the ability to route messages without side-effects it does not need FIND_NODE. All the messages that the node tries to send to non-existent nodes will either return to the sender or the next closest node to its right. An application keeps track of received messages and disregards messages that it had already seen. The node that receives messages destined for a non-existent node not send by itself knows that it has to assume its parents role and continues as outlined above. Should a node get the message that it itself has sent to one of its children, it can immediately deduce that there can be no more nodes in that subtree because all nodes in a subtree share the characteristic of having the same next bit; thus *any* of these nodes would have been closer to any other.

A KMR only *looks* binary in a certain representation, but on the other hand, it provides a strong upper-bound on its *arity* for each node, whereas KBTs/FTTs do not. This *arity* is the width of the ID-space - 1 for the root of the tree and decreases by 1 at each level. The first approach previously outlined has the ability to find existent nodes and will thus only send necessary messages. The second approach has to blindly send the maximum number of messages, many of which will end up at the same node. As soon as the next node down the chain determines all those nodes that cannot be there, this cuts down the number of messages that need to be sent. It still has to send messages to all of its parent's children that are to its right.

2.2.2 The stony way up

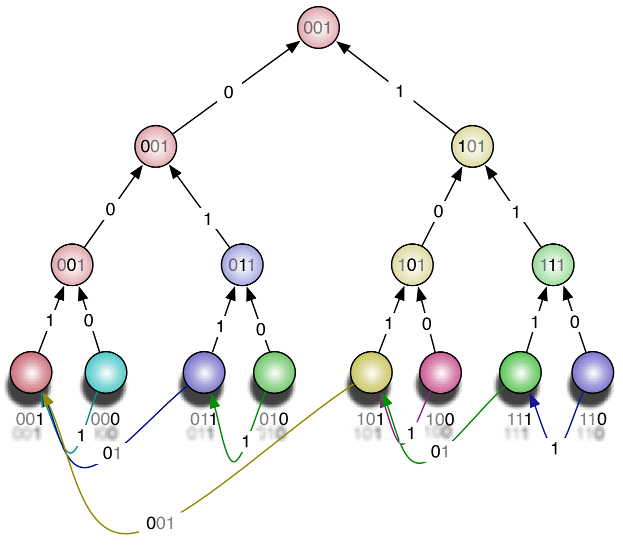

Figure 5: Graphic explanation of the aggregation phase of a KMR rooted at Key(001)

The aggregation phase works exactly opposite to the dissemination phase. To work the way up the tree, every node only needs to send one message but has to wait for as many messages as it sent in the dissemination phase to arrive.

An estimate of the number of messages can be made using the global number of nodes as a statistical hint, if that number is unknown using the distance to the root as a rough estimate, or by using the fingertable. Then, each node sends one message to its immediate parent in the tree according to the following algorithm.

Listing 2: Algorithm for the aggregation phase of a KMR

```
route( OwnID ^ (1 << len(suffix) );
```

In a KMR routing "up" the tree is done by fixing both pre- and suffix and flipping bits in the suffix "from the rear," i.e. adding bits to the front of the matching suffix.

Since, as already mentioned above, the ID-space will most likely not be fully populated, this algorithm will again end up trying to route a lot of messages to non-existent nodes. This can be avoided in the following way.

A naive way is to remember where the message in the dissemination phase came from and just assume that the parent is still good while aggregating.

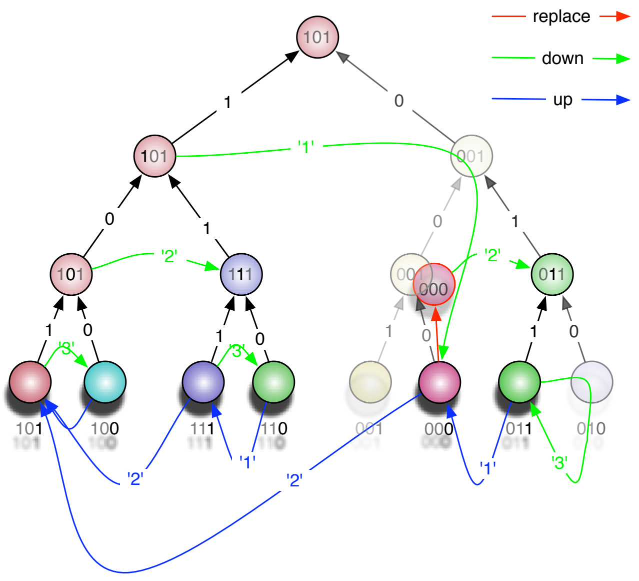

Figure 6: Summary of the both the aggregation phase and the dissemination phase of a KMR rooted at Key(101)

3 Acronyms

CDN	Content Distribution Network
DHT	Distributed Hash Table (see **??**)
FTT	FingerTable-based Tree (see **??**)
KBR	Key-Based Routing layer (see **??**)
KBT	Key-Based Tree (see **??**)
KMR	Key-based MapReduce (see **??**)
P2P	Peer-to-Peer
PWHN	PlanetenWachHundNetz

References

[1] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004. 3

[2] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sept. 2003. URL citeseer.ist.psu.edu/huebsch03querying.html. 6

[3] R. Huebsch, B. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of PIER: an internet-scale query processor. In *The Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005. URL citeseer.ist.psu.edu/huebsch05architecture.html. 6

[4] PIER. http://pier.cs.berkeley.edu/. Valid on 13.9.2006. 6

[5] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997. URL citeseer.ist.psu.edu/plaxton97accessing.html. 3, 4

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, University of California, Berkeley, Berkeley, CA, 2000. URL citeseer.ist.psu.edu/ratnasamy02scalable.html. 1

[7] R. van Renesse and A. Bozdog. Willow: DHT, aggregation, and publish/-subscribe in one protocol. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2004. URL citeseer.ist.psu.edu/642279.html. 4

[8] P. Yalagandula and M. Dahlin. A scalable distributed information management system, 2003. URL citeseer.ist.psu.edu/yalagandula03scalable.html. 6

[9] Z. Zhang, S. Shi, and J. Zhu. Somo: Self-organized metadata overlay for resource management in p2p DHT. In *Proc. of the Second Int. Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Feb. 2003. URL citeseer.ist.psu.edu/zhang03somo.html. 4