APPROVAL SHEET

Title of Thesis: DistribNet –
A Global Peer-to-Peer Internet File System

Name of Candidate: Kevin Atkinson
Master of Science, 2005

Thesis and Abstract Approved: ____________________________
Charles Nicholas
Professor
Department of Computer Science
and Electrical Engineering

Date Approved: __________________________________

Name: Kevin Jay Atkinson.

Permanent Address: 7962 Helmart Drive
Laurel, MD 20723

Degree and date to be conferred: Master of Science, 2005.

Date of Birth: May 4, 1977.

Place of Birth: Media, PA.

Secondary education:
Sandy Spring Friends School, Sandy Spring, MD, June 1996

Collegiate institutions attended:
1996-1998, West Virginia Wesleyan, Buckhannon, WV
1998-2000, University of Maryland, College Park, B.S., May 2000
2000-2005, University of Maryland Baltimore County, M.S., December 2005

Major: Computer Science.

ABSTRACT

Title of Thesis: DistribNet –
A Global Peer-to-Peer Internet File System

Kevin Atkinson, Master of Science, 2005

Thesis directed by: Charles Nicholas
Professor
Department of Computer Science
and Electrical Engineering

Most peer-to-peer networks focus on distributing documents which are currently popular, and not the long term archival of valuable documents. DistribNet, a global peer-to-peer Internet file system into which anyone can tap or add content, is different as it focuses on the long term availability of documents.

DistribNet consists of two essentially independent parts. The first part concentrates on the routing of keys. The routing strategy used is a unique combination of Pastry's routing table and Kademlia's XOR based metric; it has nearly all the same benefits of Pastry while maintaining most of the simplicity of Kademlia's XOR based strategy. The second part focuses on the actual distribution of content.

DISTRIBNET – A GLOBAL PEER-TO-PEER INTERNET FILE SYSTEM

by
Kevin Atkinson
kevin@atkinson.dhs.org

Thesis submitted to the Faculty of the Graduate School of the University of Maryland, Baltimore County in partial fulfillment of the requirements for the degree of
Master of Science
2005

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The primary goal of developing DistribNet was to prove that it is possible to develop a system which anyone can tap into or add content to, and that focuses on the long term availability of documents.

In order to build a system which addresses these I had to implement some sort of Distributed Hash Table (DHT). This in itself turned out to be a rather interesting problem, and is the area where I spent most of the time.

The primary goal is realized to some extent. However, the current system lacks several key features, most notable the ability to remove documents when space runs out.

The main contribution to this paper is in the combining of Pastry's [18] routing table and Kademlia's [12, 13] XOR based metric. This gives the system nearly all the same benefits of Pastry while maintaining most of the simplicity of the Kademlia XOR based strategy.

1.2 Problems Addressed

This theses addresses the problem of developing a system which anyone can tap into or add content to. Once a document is added to DistribNet it should remain available until it is no longer needed now, or in the future. The ability to publish content should not depend on the ability of finding someone to host it. The user simply submits the document into the system and is available for anyone to retrieve. Since a dedicated host is not required the identity of the poster need not be revealed.

In order to realize this it is necessary to implement some sort of Distributed Hash Table (DHT) in order to be able to find content without requiring a dedicated host. This thesis also addresses the problem of developing an efficient DHT. A efficient DHT should be scalable, self-organizing, and maintain good locality proprieties. It should be scalable in the sense that the worst case performance should be logarithmic to the number of nodes on the network. It should be self organizing in the sense that it automatically adopts to the arrival, departure and failure of nodes. Finally it should maintain good locality proprieties in that when routing a message it should chose nodes which are close network wise.

1.3 Related Work

1.3.1 Other File Sharing Systems

Over the last several years many types of file sharing systems have developed. The most popular ones can be grouped into three categories:

P2P File Sharing Those that focuses on allowing users to easily share files. These networks index content but don't actually store the data on the network. Instead they simply point to the location which it can be found, which is generally on the users' hard drive. Examples of these type of networks include the original Napster, Gnutella and the FastTrack network (used by Kazaa and others) [14, 7, 4].

Anonymous File Sharing Those that focus on anonymity. These networks distribute the actual content on the network, in a manner which the end-user has little control over. These networks include a number of safeguards in order to ensure that the user's identity is kept secret. Such networks include Freenet and GNUNet [1, 6].

Distributed Downloading Those that focus only on efficiently distributing a specific document. The location of the document must be provided via out-of-band means. The primary example of this is BitTorrent [2].

However, none of these focus on the long term availability of documents. *P2P File Sharing* systems only index documents, which means that files are only available as long as someone is willing to share it. Absolutely no guarantee is made of the availability of a documents over time. The very design of most *Anonymous File Sharing* systems means that unpopular documents will fall off the system. Finally, *Distributed Downloading* systems only focus on saving bandwidth; the complete document in such a system still must be made available by some on as with a normal P2P File Sharing Systems. DistribNet's design is different from all these systems since it is designed to hold onto documents indefinitely while at the same time allowing anyone to add content to the system.

Although not really considered a file sharing network per-se the web can also be used to distribute content. However, anyone who wishes to contribute content to the web must find a means to host it, which in general costs money. One of the goals of DistribNet is to allow anyone to add content without having to host it.

1.3.2 Other Distributed Hash Tables

Just as there are many different file sharing networks, there are also a large number of DHTs.

One of the most well known DHT is Chord [19, 20]. Another significance one is Pastry which DistribNet's DHT is based partly one. A lesser known DHT, which is similar to Pastry in many ways, is Kademlia [12, 13] which DistribNet is also based on. A detailed analysis of how DistribNet related these DHTs is given in Chapter 3.

Other DHTs include: PRR [15], Tapestry [21], CAN [17, 16], Viceroy [11], and a randomized algorithm due to Kleinberg [10].

1.4 Organization

The rest of this document is organized as follows:

Chapter 2 gives a general overview of DistribNet and a coarse description of how DistribNet performs the basic functions. The DistribNet routing strategy is then laid out in detail. Chapter 4 discusses the management of keys in DistribNet. Miscellaneous issues not discussed elsewhere, including how DistribNet's design focuses on the long term availability of documents, are then brought up in Chapter 5. The next chapter addresses the issue of evaluation of DistribNet. Chapter 7 then goes on to discussion limitations of the current system and possible ways to improve DistribNet in the future. Finally a conclusion is given in Chapter 8.

Chapter 2

DistribNet Overview

DistribNet consists of two essentially independent parts. The first part concentrates on the indexing and routing of keys. The routing strategy used is a unique combination of Pastry's routing table [18] and Kademlia's XOR based metric [12]; it has nearly all the same benefits of Pastry while maintaining most of the simplicity of the Kademlia XOR based strategy. This is the most well developed and tested part of DistribNet and where this paper will spend the most time on. The second, less developed part, focuses on the actual distribution of the content itself.

2.1 Architecture

DistribNet separates the indexing of documents and the actual retrieval of documents. This is different from anonymous file sharing networks, such as Freenet, which route the actual contents. However, this is similar to what most P2P file sharing networks do, such as Napster [14], Gnutella [7] and FastTrack [4]. However, unlike these P2P examples, DistribNet also stores the actual content on the network.

The first part of DistribNet involves the storing of *routed* keys. The second part involves the storage of *non-routed* or data keys.

Routed keys are distributed based on a distributed hashing algorithm so that the keys can always be found. Routed keys are small (under 1k) and will generally, but not always, point to the location of larger data. Non-routed keys can be distributed in any fashion. They will typically be stored where it will be the most beneficial in terms of performance and availability. Routed keys are generally used to point to other data. They are stored as appendable lists so that they can be updated. Routed keys that are used to locate data keys are indexed based on the hash of the data key they point to. The primary type of routed key in DistribNet is the Index key, which is described in more detail in Chapter 4.1.

Non-routed, or data, keys are immutable and always indexed based on the SHA-1 hash of the content. Depending on the location of the data it can either be considered a Permanent key or a Cached key. Cached keys are freely deleted when space is needed while permanent keys are only deleted if the data is also available somewhere else on the network. A cached key can become a permanent key if necessary to ensure that data does not disappear from the network. See Chapter 4.2 for more details on the data key.

Nodes in DistribNet are uniquely identified by the 160-bit SHA-1 hash of the public key. Since SHA-1 hashes are used the nodes will be evenly distributed.

2.2 Adding Data

The actual process of uploading data to DistribNet is fairly involved, but here is a simplified version of it. The SHA-1 hash of the content is computed. Then several nodes are chosen as candidates to receive the data. The data is then uploaded to those nodes. If one of the nodes refuses to accept the data another node is chosen. The final location of the data item is then stored in a routed key which is then uploaded. A more detailed explanation of how data is stored in DistribNet can be found in Chapter 4.2.1.

2.3 Retrieving Data

The actual process of retrieving data from DistribNet is again fairly involved, but here is a simplified version of it. In order to retrieve data the SHA-1 hash of the data's content must be known. The location of the data is located by retrieving a routed key which is based on the data's hash. This key will contain a list of nodes that might have the data. The closest node is chosen from that list. The data is then downloaded from that node. If that node does not have the data the next node on the list is tried. If none of the nodes on the list have the data the request will fail. A more detailed explanation of how data is retrieved in DistribNet can be found in Chapter 4.2.3.

2.4 Limitations

There is currently no way to retrieve a document by name, only by the hash of the content. A future version of DistribNet will address this issue by adding a new type of key, the map key, which will be indexed based on the hash of the title and can be updated. (see Chapter 7.1).

Furthermore, the current version of DistribNet does not provide any mechanism for removing obsolete data. The network may eventually run out of space. The main reason for this is due to a lack of some sort of rule to determine when a document is truly obsolete or useless. (see Chapter 7.2).

Chapter 3

DistribNet Routing

One of the most important aspects of DistribNet is its ability to effectively route keys. The DistribNet routing protocol has nearly all the same benefits of Pastry while maintaining most of the simplicity of the Kademlia XOR based strategy. In particular it is completely decentralized, scalable, and self-organizing; it automatically adapts to the arrival, departure and failure of nodes. Like Pastry, DistribNet also has takes into account network locality to minimize the distance messages travel [18]. However, due to the use of a strictly XOR based metric as used in Kademlia, DistribNet routing, is simpler, as a single routing algorithm is used from start to finish [12, 13].

3.1 Overview

DistribNet uses a distributed hash table (DHT) in order to locate keys. Unlike a regular hash table which computes a hash and then uses direct lookup to locate the key, a DHT computes the hash and then uses it to find the node closest to the key based on some metric. The node receiving the key will then again look for the node closest to the key. This process, known as routing, will continue until either the key or the closest node is found. This process works because each node keeps track of different neighboring nodes. The number of other nodes any particular node keeps track of is only a small fraction of the total number of nodes on the network. Many different routing algorithms have been developed in recent years. Most of these algorithms can find the desired key in logarithmic time, however they differ greatly in the flexibility in choosing neighbors. This flexibility plays an important role in how well the algorithms perform when nodes go offline and how well it can adapt to underlying Internet topology. The underlying geometry of the network plays a key role in the flexibility the DHT algorithm. Only the ones which offer good flexibility are mentioned here. For others please see the paper "The impact of SHT Routing Geometry on Resilience and Proximity."[8].

One well known routing algorithm is Chord [19, 20]. In Chord nodes are arranged in a ring like structure. The distance between two nodes A and B is the clockwise numerical distance between A and B on the circle. Mathematically this is essentially:

$$d(a,b) = a - b \pmod N$$

where a and b are the keys for node A and B respectively and N is the size of the network. In order to route a key to to the closest node, Chord maintains a finger table where the i^{th}entry in the table is the node closest to $a+2^i$ on the circle. Thus, a node can route a key in $logN$ hops since each hop cuts the distance in half. Although the original Chord algorithm uses a specific set of neighbors in the finger table, this rigid selection of neighbors is in no way fundamental to the ring geometry. Thus the Chord algorithm can fairly easily be modified to support support flexible node selection. The main drawback of Chord is that, while it may be easy to describe, it is tricky to implement. In particular the maintenance of the finger table is tricky.

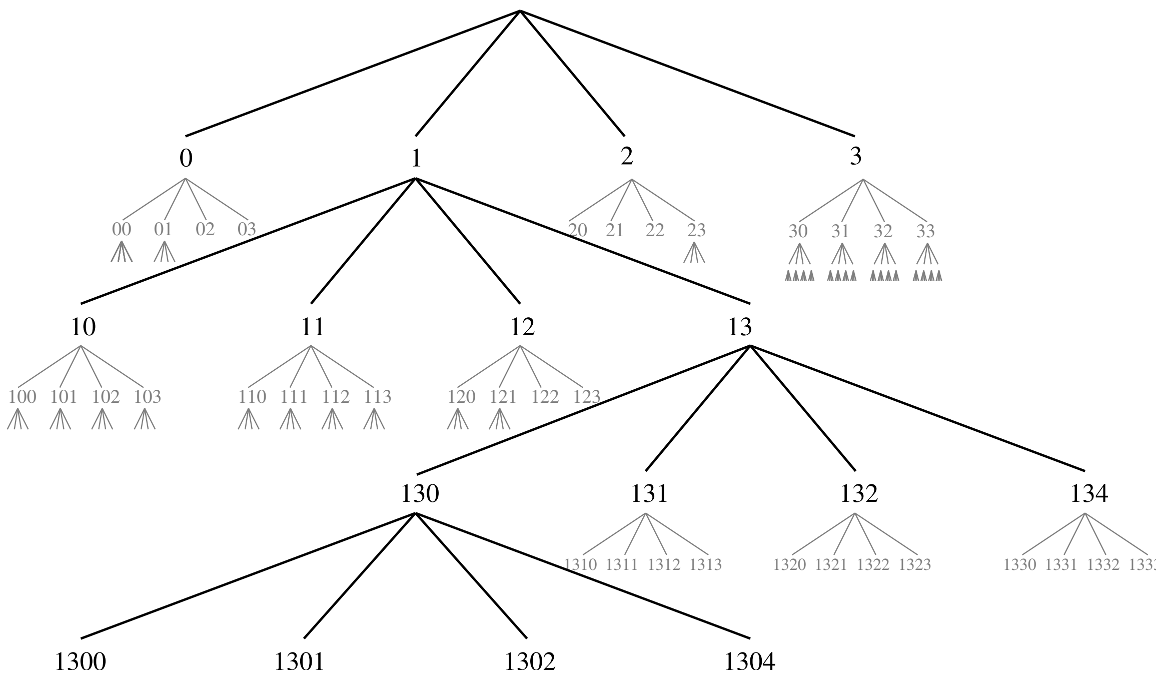

Figure 3.1: A tree based geometry as seen by node 1300 with digits of base 4. The large entries are the part of the network that node 1300 needs to keep track of. The smaller, shaded, entries are the other nodes in the network.

Another way to implement a DHT routing algorithm is to base it on a tree based geometry. In a tree based geometry nodes are distributed evenly throughout the tree based on their keys to form a essentially balanced tree. To do this each key is thought of as a serious of digits and branching takes place based on these digits as shown in Figure 3.1. The "distance" between two nodes is the height of the smallest common subtree. The trick is to store the right part of the tree in each node so that each hop will decrease the distance by at least one. The maximum number of hops is then the height of the tree. A strategy to maintain the right part of the tree, known as the routing table, will be discussed in the next section. Describing a routing algorithm may be tricky, but the structure of the routing table leads to easy maintenance. The tree geometry can have very flexible node selection if the routing table keeps track of multiple nodes for each possible branch.

Pastry [18], another popular routing algorithm, uses a tree based geometry but switches to using numerical distance for the last couple of hops which can be described mathematically as:

$$d(x,y) = |x - y|$$

This is necessary because it is not possible to uniquely find the closest node based solely on the tree based method. For example the keys 1234, 1252, and 1278 are all the same distance from each other since they all have the same common subtree. However, numerically they are at different distances from each other. Strictly speaking it possible to have two nodes, say 1232 and 1238 be the same distance from a key, say 1235, but in practice this is extremely unlikely to happen because the number of possible keys is generally much larger than the number of actual nodes on the network.

Another routing algorithm, Kademlia [12], avoids having to use two different metrics by using a novel routing metric – the bitwise exclusive or (XOR) – which can be described mathematically as:

$$d(x,y) = x \oplus y$$

For example the keys 0xF0 and 0x1F will have a distance of 0xDF . The XOR metric metric forms a valid metric space. To see that, note that it satisfies the basic axioms of a metric space:

$$\begin{aligned} \forall x,y \geq 0: \quad & d(x,x) = 0 \\ & d(x,y) \geq 0 \\ & d(x,y) = 0 \mapsto x = y \\ & d(x,y) = d(y,x) \end{aligned}$$

as well as the triangle inequality:

$$\begin{aligned} \forall x,y,z \geq 0: \qquad & x + y \geq x \oplus y \\ \therefore \quad & d(x,y) + d(y,z) \geq d(x,y) \oplus d(y,z) = d(x,z) \\ \therefore \quad & d(x,y) + d(y,z) \geq d(x,z) \end{aligned}$$

A metric space is an abstraction of the common notion of distance. Forming a valid metric space is significant since many of the concepts used for numerical distance in Euclidean space, the most well known metric space, can also be used with other metric spaces such as the XOR metric.

The XOR metric also directly relates to the metric used in the tree based routing algorithm, when the base B of the digits is a power of 2, via the formula:

$$\lfloor \log_B (x \oplus y) \rfloor$$

Furthermore, this metric can also be used to uniquely locate a closest node since, like the ring metric used in Chord, it is *unidirectional*, that is, for any given point x and positive distance δ there is exactly one point y such that $d(y,x) = \delta$. As already noted above the simple numerical distance does not have this property as for every point y on the number line there are exactly two points that have the distance δ in particular $y - \delta$ and $y + \delta$.

The unidirectional property also ensures that all lookups for the same key converge along the same path, regardless of the originating node. This property is useful since it makes caching of commonly accessed keys as easy as replicating them along this common lookup path.

The routing algorithm that DistribNet uses combines pieces of both Pastry and Kademlia. The routing table closely resembles that of Pastry, but, DistribNet uses a strict XOR metric much like Kademlia does. Using a strict XOR metric simplifies the routing and also avoids some of the problems Pastry has. One of these problems is that nodes close by the first metric, the tree based one, can be quite far by the second, the numeric one. For example 0x0FFF and 0x1000 which are very far away by the first one, as they have only one common subtree, are very close by the second one. This creates discontinuities for particular node keys, reduces performance, and makes formal reasoning about the worst-case behavior difficult, as observed by Maymounkov and Mazières [12]. In the end the DistribNet routing table ends up being very similar to what Kademlia uses but with one important difference which will be discussed in Chapter 3.2.1.‘

3.2 Routing Table

In order to to be able to locate keys in a DHT some sort of routing table needs to be maintained. DistribNet routing table is very similar to Pastry's. In Pastry keys are seen as a sequence of digits with base 2^b. The routing table is essentially a matrix with at most $\frac{k}{b}$ rows, where k is the number of bits in the key, and 2^b columns as shown in Figure 3.2. Each row maintains a list of other nodes such that the keys of those nodes have the property that the first N digits are the same as the current node's key, where N is the row number which starts with 0. The node's within a row are grouped into columns such that the $N+1$ digit of the nodes key is the same as the column number, which also starts with 0. Within each row one column is empty. The empty column is the column in which the $N+1$ digit is the same as the current node. All the entries in the next row technically fit in this column.

In DistribNet the base chosen is $2^4 = 16$ and the number of rows is fixed at 8. Four was chosen as the base size for several reasons 1) it is a power of two, 2) when keys are thought of as a sequence of digits a base size of 4 means that the digits will be hexadecimal, 3) the Pastry paper hinted that 4 would be a good choice. The number of rows was chosen to be large enough so that there is no possibility that the last row will be used when dealing with a moderate size network during testing. Theoretically, the number of rows does not need to be fixed and it can change based on the network size. It was fixed in the initial implementation for simplicity.

Unlike Pastry there is no real leaf set. Instead the "leaf set" consists of all rows which

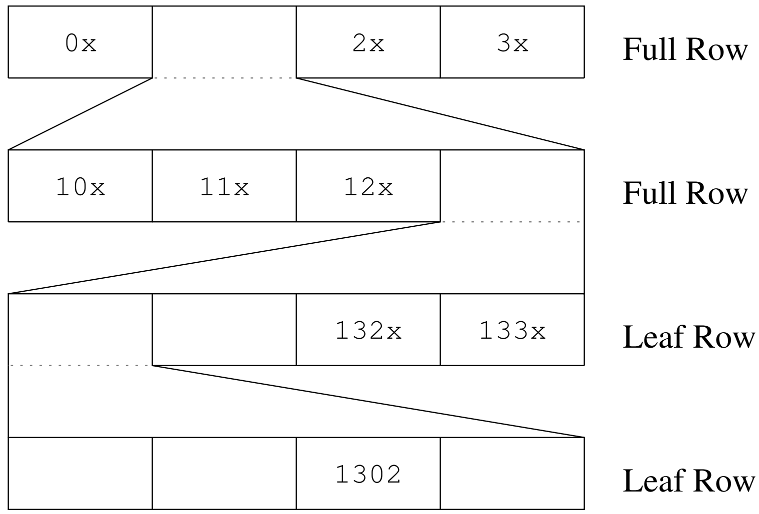

Figure 3.2: DistribNet routing table for the node 1300, with $b = 2$. All numbers are in base 4, x represents an arbitrary suffix.

are not “full”. A full row is a row which contains 15 full entries with the extra empty entry being the one which represents the common digit with the node’s key, and thus will never be used. Not having a true “leaf set” simplifies the implementation since a separate list does not need to be maintained and the routing algorithm remains the same instead of switching to numerical distance as Pastry does. This also means that all the nodes in the leaf set will maintain the same set.

A row is considered full in DistribNet if 15 of the 16 entries are full in the current node *and* other nodes on that row also have 15 of the 16 entries full. For each full row DistribNet will try to maintain at least two nodes for each entry. This way if one node goes down the other one can be used without affecting performance. When a node is determined to be down, as opposed to being momentarily offline, DistribNet will try to replace it with another node that is up. With this arrangement it is extremely likely that at least one of the two nodes will be available. A full row can become a leaf row if the entry count drops below 15.

For each incomplete row DistribNet will attempt to maintain as many nodes as are available for that entry so that every other node in the leaf set is accounted for. From time to time DistribNet will contact another node in the leaf set and synchronize its leaf set with it. This is possible because all nodes in the leaf set will have the same set. Down nodes in the leaf set will be removed, but the criteria for when a node is down for a leaf set is stricter than the criteria for a full row. If a leaf row becomes full then excess nodes will be removed.

Unfortunately on a dynamic network where nodes are constantly coming and leaving the current strategy of distinguishing between a full and leaf row can lead to an oscillation between the two states. If this happens network bandwidth will be wasted as DistribNet will constantly be adding and removing nodes from the routing table. However, the rate which this happens should be low enough that it does not waste a significant amount of bandwidth.

3.2.1 Comparison to Kademlia

The routing table DistribNet uses is based on Pastry. However, it also turns out to be very similar to Kademlia's with one key difference.

Kademlia routing table is described in [13] (Chapter 2.4) as follows (*u* simply referrers to the current node):

> The routing table is a binary tree whose leaves are *k*-buckets. Each *k*-bucket contains nodes with some common prefex of their IDs. The prefix is the *k*-bucket s position in the binary tree. Thus, each *k*-bucket covers some range of the ID space, and together the *k*-buckets cover the entire 160-bit ID space with no overlap.
>
> Nodes in the routing tree are allocated dynamically, as needed . . . Initially,

> a node u's routing tree has a single node—one k-bucket covering the entire ID space. When u learns of a new contact, it attempts to insert the contact in the appropriate k-bucket. If that bucket is not full, the new contact is simply inserted. Otherwise, if the k-bucket's range includes u's own node ID, then the bucket is split into two new buckets, the old contents divided between the two, and the insertion attempt repeated. If a k-bucket with a different range is full, the new contact is simply dropped.

Kademlia's k-bucket is very similar to a row in DistribNet's table. However, a DistribNet row doesn't need to be split as a k-bucket does in Kademlia. The place where Kademlia will split a k-bucket corresponds to the empty column in DistribNet. Thus, in a way, a DistribNet row is split when the table is created rather than dynamically, as needed. Since the table won't get very large this is not an issue in practice.

So, even though DistribNet and Kademlia's routing table are implemented differently they end up being functionally very similar. However there is one key difference. This difference comes from the fact that Kademlia has no concept of full or leaf rows. DistribNet tries very hard to make sure it knows about every other node on the network that has the same first n digits as its own node key where n is defined as the depth of the first leaf row while Kademlia makes no such effort.

Keeping track of every node in the leaf set allows DistribNet to get a very accurate estimate on the size of the network. If the network is static (i.e. nodes are not joining and leaving the network) the estimate will be exact. Otherwise it should be very close. Such an estimate is only possible if, after a certain point, every node on the network with the same prefix is accounted for. Thus such as estimate isn't possible with Kademlia. See Chapter 3.4 for the details on how the network size is estimated.

However, as previously mentioned, the current manner by which DistribNet distinguishes between a full and leaf row can lead to a constant adding and removing of nodes from the table. Kademlia does not have this problem, since it is specific to maintaining a leaf set.

3.3 Routing A Message

Routing in DistribNet is very simple. Whenever a route request is sent, DistribNet looks for the closest node to the key by using the XOR metric and then forwards the request to that node. This is done recursively until the current node is the closest node on the network, in which case the search terminates. Since the XOR metric is unidirectional there will always be exactly one node in the routing table that is the closest, it is never the case that two nodes will be the same distance from a key. Furthermore, the process is guaranteed to terminate since each routing step involves finding a closer node and will stop if a closer node cannot be found.

Under ideal conditions each step matches another upper order digit, or when viewed as a tree decreases the heigt of the common subtree by at least one. Therefore, under ideal conditions the number of routing steps is:

$$\lceil \log_{16} N \rceil$$

The expected number of routing steps is very close to ideal since DistribNet tries very hard to maintain ideal conditions. The absolute worst case performance is N since each routing step involves finding a node that is at least a little close than the current node. However, the chances of the network degrading to the worst case is next to nothing.

3.4 Size of Network

One very nice property of using a tree based geometry is that is possible to theoretically calculate the exact size of the netwoek via the formula:

$$S(k) = \begin{cases} l(k) & \text{if } l(k) \text{ exists} \\ \sum_{i=0}^{B-1} S(k:i) & \text{otherwise} \end{cases}$$

where k is a string of digits of base B (16 in the case of DistribNet), $k:i$ is the string k with the digit i appended to the end. $l(k)$ is said to exist when all node keys starting with k have the same leaf set, in which case $l(k)$ is the size of this leaf set. The size of the network is then $S(\varepsilon)$ where ε represents the empty string.

Of course, on a live network, where nodes are constantly coming and going, the exact size can not be calculated. However, we can get very close. In addition it is possible to get an idea of how accurate this estimate is.

Each node on the network maintains an estimate of S for the empty string, for the first digit of its node key, for the first two digits, and so on. Let $S_i(k)$ be equal to $S(k)$ for the first i digits of k. Let $S_i$represent $S_i(K)$ where K is the current node's key. Let $N_{i,j}$be the set of nodes in row i and column j of the current node's routing table. Finally let J_i be the set of integers between 0 and $B-1$ except the one which is equal to the $i+1$ digit of K. Then:

$$\begin{aligned} S &= S_0 \\ S_i &= \begin{cases} \sum_{j \in J_i} S_{i,j} + S_{i+1} & \text{full row} \\ L_i & \text{leaf row} \end{cases} \\ S_{i,j} &= \operatorname*{avg}_{n \in N_{i,j}} \left(S_{i+1}(n)\right) \end{aligned}$$

for each full row. L_i is the number of nodes in a leaf row which is known since DistribNet keeps track of every node on the network that is in the leaf set.

Each node on the network also maintains an estimate of the accuracy of S which is a number between 0 and 1.0. The accuracy A is calculated as follows:

$$
\begin{aligned}
A &= A_0 \\
A_i &= \begin{cases} \dfrac{\sum_{j \in J_i} A_{i,j} + A_{i+1}}{B} & \text{full row} \\ \dfrac{A_i^- + 2 - \left| \frac{S_i^- - S_i}{S_i^- + S_i} \right|}{3} & \text{leaf row} \end{cases} \\
A_{i,j} &= \operatorname*{avg}_{n \in N_{i,j}} \left(A_{i+1}(n)\right) \frac{\operatorname*{avg}_{n \in N_{i,j}} \left(S_{i+1}(n)\right)}{\max_{n \in N_{i,j}} \left(S_{i+1}(n)\right)}
\end{aligned}
$$

where S^- and A^- represent the value of S and A from the previous iteration with an initial value of 0. The value of A does not have an exact meaning except that higher values represent a more accurate estimate. The maximum value of A, 1.0, can never be reached, but on a stable network the value of A can become very close to 1.0.

The relevant values of $S_i(n)$ and $A_i(n)$ for nodes other than the current node is stored with the node in the routing table and is updated from time to time.

Being able to get a precise estimate of the network size is useful in itself. However, this information is also used by the DistribNet algorithm to distinguish between a Full row and a Leaf row, which is a key part of DistribNet routing.

3.5 Table Maintenance Details

There are four messages related to table maintenance: ROUTING-INFO-REQ, ROUTING-INFO, ROUTING-COUNT-REQ, and ROUTING-COUNT.

A ROUTING-INFO-REQ is sent in order to retrieve another node's routing table. A ROUTING-INFO-REQ contains one parameter which is a node key. The message is routed until the closest node to the key is found, excluding the key itself. This message is most commonly sent when a node first joins the network. The request is sent in order to discover the node which is closest to itself (via the XOR metric). The response to the message is a ROUTING-INFO message which simply contains a list of all nodes in the routing table. The ROUTING-INFO message is also sent by itself from time to time to make sure all the nodes in the leaf-set know about each other.

A ROUTING-COUNT-REQ has two purposes, to make sure a node is still alive, and to maintain an accurate estimate on the number of nodes on the network. Unlike the ROUTING-INFO-REQ message it is sent to a specific node and is not routed. The response to this message is a ROUTING-COUNT message. A ROUTING-COUNT message contains the node's values of S_i, A_i and C_i where i goes from 0 to the number of rows in the routing table (currently eight). S_i and A_i are as described in Chapter 3.4 and C_i is a count of the number of columns in each row that have at least one node in them. DistribNet uses this count to determine if a row is a full.

3.5.1 Joining The Network

In order to join the network the node must have at least one other node in the routing table. The procedure for joining the network for the first time is the same as the procedure for rejoining the network after being disconnected for a while.

When the node first joins a network it will send a ROUTING-INFO-REQ message and then update its routing table with the information returned which is the routing table of the closest node on the network. Next the node will perform table maintenance for the first time. Finally, the node will send a ROUTING-INFO message to every other node in its leaf set to announce its presence.

3.5.2 Routine Table Maintenance

When a node first joins a network, and then at fixed intervals T table maintenance will be performed in order to to keep the routing table current.

Updating Route Count Information

The first step performed in table maintenance is to update the count information for selected nodes in the routing table. This is done using the following process.

1. Flag all nodes that have not been updated in T_R time which is a number generally several times larger than T.
2. Of the flagged nodes randomly select at most $P + \left\lceil R\frac{T}{T_R} \right\rceil$, where P is the *carryover* value from the previous iteration and R is the number of nodes in the routing table, to be updated. This is done so that during each maintenance interval roughly the same number of nodes will be updated.
3. Reset the carryover value P to 0.
4. Send a ROUTING-COUNT-REQ to each of the selected nodes and wait for them to respond with a ROUTING-COUNT message. If a node does not respond then increment the carryover value P. If the node is online then the relevant values of S_i, A_i and C_i for that node are updated.

During the first maintaince interval, i.e. when a node first joins the network, all of the nodes in its table will be flagged in step 1. Assuming ideal conditions, that is T_R is a multiple of T and R is a multiple of T_R, step 2 will then select $R\frac{T}{T_R}$ nodes to update. During the next interval, step 1 will flag $R - R\frac{T}{T_R}$ nodes, of which another $R\frac{T}{T_R}$ will be selected. During the ith interval, step one will flag $R - iR\frac{T}{T_R}$ nodes until the $2\frac{T_R}{T}$ interval. From that point on, exactly $R\frac{T}{T_R}$ nodes will be flagged in each interval. Furthermore, each $i\frac{T_R}{T} + n$ interval, for all i with $n \in \left[0, \frac{T_R}{T}\right)$, will flag the exact same nodes. This of course assumes that every node is online and that there are not any new nodes joining the network.

If a node does not respond in step 4 then that node will be flagged again during the next interval in step 1. Thus there is a chance it will be selected again during step 2. If it doesn't get selected during this interval then there is a chance it will get selected in the next one, and so on, until it is finally selected, which will be within $\frac{T_R}{T}$ maintenance intervals because of the use of the carryover value P. If the node went offline after the network had been up a while (i.e. after $\frac{T_R}{T}$ maintenance intervals) then there is a high probability it will be selected again in the next maintenance interval. If this is the only node that went off line and no new nodes are added then the probability of it being selected in the next interval is 100%.

On a live network with nodes coming and going this system should ensure that for every maintenance interval roughly the same number of nodes will be contacted and that every node will be contacted at least once for every $\frac{T_R}{T}$ maintenance intervals.

Updating The Network Size

The estimate of the network size is then updated which includes calculation values for S_i and A_i as described in Chapter 3.4.

Determining If A Row Is Full

Then each row is reexamined to determine if it is a full row or a leaf row. A row i is considered full only when $C_i = C_{\max}$ and $\sum f_i(n) \geq 3$ for all nodes n on the row where

$$f_i(n) = \begin{cases} 1 & \text{if } C_i(n) = C_{\max} \\ 0 & \text{if } C_i(n) \neq C_{\max} \text{ and } A_{i+1}(n) < 0.85 \\ -1 & \text{if } C_i(n) = C_{\max} - 1 \text{ and } A_{i+1}(n) \geq 0.85 \\ -\infty & \text{if } C_i(n) < C_{\max} - 1 \text{ and } A_{i+1}(n) \geq 0.85 \end{cases}$$

and $C_{\max}$ is the maximum value C_i can have which is equal to one less than the number of columns in the table. The value of A_{i+1} is used instead of A_i because that is the value that is stored with the node. If $A_{i+1} < 0.85$ then the row count (C_i) from that node should not be trusted as it probably has not been on the network that long.

Synchronizing Leaf Sets

The next step is to synchronize its leaf set with other leaf nodes to ensure that it knows about every possible node on the network that should be in its leaf set.

Repairing

When there are less than two nodes for a column in a full row DistribNet will attempt to locate another node for that position.

3.5.3 Leaving The Network

No special action is taken to leave a network. A node simply disconnects. Other nodes will eventually discover that the node went offline and take appropriate action.

Chapter 4

Key Management

The previous chapter discussed how DistribNet routes a message in order to be able to find the node with the information, or key, of interest, however it did not discuss how the actual keys are managed. This chapter will address the issue. Also addressed will be how DistribNet manages content, that is the data to which the keys point.

There are two primary key types in DistribNet: Index and Data Keys. Index keys in DistribNet are routed keys which point to other data, which is generally a Data Key. Data Keys on the other hand are non-routed keys which point to where the content can be downloaded from.

4.1 Index Keys

Index keys in DistribNet are routed keys which point to other data, in particular Data Keys. They are essentially appendable lists. They are designed so that, with a little modification, they can also be used as generic appendable lists.

Each index key contains a list of nodes on the network which should contain the corresponding data key. At any given time at least X index keys are stored on the closest X nodes via the XOR metric. Index keys stored this way are known as *permanent index keys*.

Since the size of index keys is small they will generally be highly replicated. However, the replicas will *not* be identical as maintaining a complete list of all the nodes which contain a popular key in every replicate is simply not practical. Instead each replica only contains a list of a few close by nodes (via some metric, see 5.2) which currently have the data key in their cache and the list of nodes which have the data key in the permanent storage.

Index keys are cached along the path of retrieval. Because the XOR metric is unidirectional lookups for the same key converge along the same path. Thus a cached index key is likely to be seen by future lookups for the same key by other nodes before the permanent index key is found, thus reducing the number of hops required to locate the key. The more popular a key is, the higher the degree of replication and thus the soon it will be found. [12]

4.2 Data Keys

Data keys is DistribNet are keys which are indexed based on their content and hold an arbitrary amount of non-mutable data. Since they are large (up to around 32K) they will generally not be routed.

4.2.1 Details

Data keys will be stored in maximum size blocks of just under 32K. If an object is larger than 32K it will be broken down into smaller size chunks and an index block, also with a

Data Block Size:	$2^{15} - 128 = 32640$
Index block header size:	40
Maximum number of keys per index block:	1630
Key Size:	20

Maximum object sizes:

direct	$\Rightarrow$	$2^{14.99}$	about 31.9 kilo
1 level	$\Rightarrow$	$2^{25.66}$	about 50.7 megs
2 levels	$\Rightarrow$	$2^{36.34}$	about 80.8 gigs
3 levels	$\Rightarrow$	$2^{47.01}$	about 129 tera
4 levels	$\Rightarrow$	$2^{57.68}$	
5 levels	$\Rightarrow$	$2^{68.35}$	

Figure 4.1: Data Key Details

maximum size of about 32K, will be created so that the final object can be reassembled. If an object is too big to be indexed by one index block the index blocks themselves will be split up. This can be done as many times as necessary therefore providing the ability to store files of arbitrary size. DistribNet will use 64 bit integers to store the file size therefore supporting file sizes up to $2^{64} - 1$ bytes.

Data keys will be retrieved by blocks rather then all at once. When a client first requests a data key that is too large to fit in a block an index block will be returned. It is then up the client to figure out how to retrieve the individual blocks.

Please note that even though blocks are retrieved individually they are not treated as truly independent keys by the nodes. For example a node can be asked which blocks it has based on a given index block rather then having to ask for each and every data block. Also, nodes maintain persistent connections so that blocks can be retrieved one after another without having to re-establish to connection each time.

Data and index blocks will be indexed based on the SHA-1 hash of their contents. The exact numbers are shown in Figure 4.1.

Why 32640?

A block size of just under 32K was chosen because I wanted a size which will allow most text files to fit in one block, most other files with one level of indexing, and just about anything anybody would think of transferring on a public network in two levels and 32K worked out perfectly. 32640 rather then exactly 32K was chosen to allow some additional information to be transferred with the block without pushing the total size over 32K. 32640 can also be stored nicely in a 16 bit integer without having to worry if it is signed or unsigned.

However, the exact block size is not fixed in stone. If, at a latter date, a different block size is deemed to be more appropriate then this number can be changed.

4.2.2 Storage

Permanent data keys will be distributed essentially randomly. However, to ensure availability the network will try to ensure at least N nodes contain the data. Nodes which are responsible for maintaining a permanent key will know about all the other nodes on the network which are also responsible for that key. From time to time it will check up on the other nodes to make sure they are still live and if less than $N-1$ other nodes are live it will look for another node to maintain a copy of the key. It will first try asking nodes which already have the key in its cache to maintain a permanent copy of the key. If this fails it will chose a random node to ask and will keep trying, selecting a different random node each time, until some node accepts or one the original nodes becomes live again. The exact value for N and how hard DistribNet tries to ensure a keys availability will be based on its estimated worth.

Cached data keys will be distributed based on where it will do the most good performance wise. For the initial implementation cached keys are simply stored on the node that downloaded them. When space runes short the oldest cached keys will be deleted. Age is based on the last time a user of the local node accessed the file. Age does not include accesses from other nodes. The theory here is that if another node accessed a key it will also store a copy of the key, thus there is no need to cache it.

4.2.3 Retrieval

When a node A wants to retrieve a key K one of two things will happen. If it has good reason to believe that a nearby node has the key it will attempt to retrieve it from that node, otherwise it will send a request to find other nodes which have the key.

To do this, node A will contact a node, B, whose key is closer to K than node A's key. Node B will in turn contact C etc, until an answer is found which for the sake of argument will be node E. Node E will then send a list of possible nodes L which contain key K directly to node A. Node E will then send the result to node D, which will send it to C, etc. Node E will also add node A to list L with probability of say 10%, Node D will do the same but with a probability of say 25%, etc. This will avoid the problem having the list L becomes extremely large for popular data but allow nodes close to A to discover that A has the data since nodes close to A will likely contact the same nodes that A tried. Since A requested the location of key K it is assumed that K will will likely download the data. If this assumption is false then node A will simply be removed from the list latter on.

Once A retrieves the list it will pick a node from the list L based on some evaluation function, let's say it picks node X. Node X will then return the key to node A. The evaluation function will take several factors into account, including distance, download speed, past reputation, and if node A even knows anything about the new node.

If node X does not send the key back to node A for whatever reason it will remove node X from the list and try again. It will also send this information to node B so it can consider removing node X from its list, it will then in turm notify node C of the information, etc. If the key is an index block it will also send information on what parts of the complete key node X has. If the key is not an index block than node a is done.

If the key is an index block then node A will start downloading the sub-blocks of key K that node X has. At the same time, if the key is large or node X does not contain all the sub-blocks of K, node X will chose another node from the list to contact, and possibly other nodes depending on the size of the file. It will then download other parts of the file from the other nodes. Which blocks are downloaded from which nodes will change based on the download speed of the nodes so that more blocks are downloaded from faster nodes and less from slower, thus allowing the data to be transferred in the least amount of time. If after contacting a certain number of nodes there are still parts of the key that are not available on any of those nodes, node A will perform a separate query for the individual blocks. However, I imagine, in practice this will rarely be necessary.

Chapter 5

Miscellaneous Issues

This chapter will discussion a few miscellaneous issues important to DistribNet that were not presented anywhere else.

5.1 Long Term Availabity

DistribNet was designed with the long term availability of documents in mind. One key design decision made towards this goal was the separation of the indexing and storing of data. This separation allows special measures to be taken to ensure that documents are always available, something that is not possible with networks that don't have this separation. In these networks data has to be stored where is can be found based on its key, which is generally some sort of hash of the data itself.

Another key design decision was the use of DHT for the indexing of documents, which allows for any document that exists on the system to be found in $\log N$ time, no mater how unpopular it is. This is in sharp contract to systems like Freenet [1], in which finding unpopular documents that exist on the system can take much longer than $\log N$. The small size of the index keys, as compared to the data itself, help to ensure that every node will be able to index all of keys that it is responsible for.

Another key factor to ensuring the long term availability of documents in DistribNet is the inability for anyone to manually remove documents from the system. Thus, users don't have to worry about a document disappearing from the network, no matter how controversial it is.

5.2 Distance determination

In order to minimize network traffic it is important to be able to estimate how far away a node is on the network so that the closest node can be selected. There are several different ways network distance can be measured:

1. The number of hops it takes to get from point A to point B.
2. The physical cost to send data from point A to point B. For example the cost of sending data over a LAN is relatively cheap compared to the cost of sending data over the Internet.
3. The latency.
4. The amount of bandwidth available.

In order to determine the best node to choose all of these factors should be considered. However it is often, but not always, sufficient to only consider the number of hops.

One very coarse estimate for node distance would be to use the XOR distance between two node's IP address since closer nodes are likely to share the same gateways and nodes really close are likely to be on the same subnet. This is what the current implementation of DistribNet does.

Another way to estimate node distance relies on the the fact that node distance, for the most part, obeys the triangle inequality. For each node in the list of candidate nodes some information about the estimated distance between that node, node E, in the list and the node storing the list is maintained by some means. For node A to estimate the distance between a node on the list, node X, and itself all it has to do is combine the distance between it and E with the distance between E and X. The combination function will depend on the aspect of distance that is being measured. For the number of hops it will simply add them, for download speed it will take the maximum, etc.

5.3 Cache Consistency

Maintaining cache consistency is a difficult problem for any network. Some networks, such as Freenet, avoid the issue all together by not having mutable keys. Other networks only keep cached copied of data around for a short time span, therefore reducing the chance of the cache data being out of date. Once a cached copy gets out of date the node either throws the copy away, or checks to see if a newer copy is available. Either approach creates unnecessary network traffic. Another approach is to have the server notify other nodes whenever the data changes. This approach will not scale well as the number of nodes a server needs to keep track of will grow with the network and is completely impractical for a large distributed network. A similar, but more scalable, approach is for nodes on the network to notify each other whenever a key changes. This is the approach that DistribNet uses.

The other issue that needs to be dealt with is conflicts. That is when two different people modify the same key at nearly the same time. DistribNet avoids this problem by only allowing keys to be added to. If two nodes add to the same key at the same than the conflict can be resolved by simply including both additions.

Chapter 6

Evaluation

I have tested DistribNet in two ways. The first is by testing the system on real PCs in a emulated network environment and the other is by running a simulation to better understand the size of the routing tables.

6.1 Tests on Real PCs

DistribNet has been tested by running up to around 16 instances on a single machine where each instance is treated as separate node. Using the Emulab testing framework I have tested DistribNet on 64 different machines each running 16 instances of DistribNet giving a total number of nodes of 1024. Emulab is a system in which you get exclusive access to real PCs for a limited amount of time.

Tests that I have performed include:

1. Making sure that all nodes can find find each other by bootstrapping each node with two other randomly chosen nodes and starting them all up at the same time. I then

check each nodes routing table to make sure it has found all of the other nodes it should have, and that they all have an accurate estimate on the number of nodes on the network.

2. Stressing the ability of nodes to eventually find each other by only bootstrapping each node with one other node so that they form a chain. For example in a three node experiment node C will only know about node B, node B will only know about node A, and A won't know about any other nodes when they first start up.

3. Once all the nodes are up, randomly killing off nodes and checking that messages are still able to be routed properly. I also check that other nodes eventually recognize that the dead nodes are down and adjust the routing tables accordingly.

4. Post content to the network and make sure that I can retrieve the data from all the other nodes on the network.

6.2 Evaluating the Size of the Routing Tables.

In order to better understand the size of the routing table I wrote a C++ program which simulates what the routing tables in DistribNet will look like. Simulations were done for network sizes up to around 16 million. In particular:

1. The simulation was run three times for network sizes of even powers of 4, from 256 (4^4) to 16,777,216 (4^{12}). Even powers of 4, rather than 2, were chosen in the interest of time as the simulation can take a very long time to run for large network sizes (a single simulation for a network size of 16,777,216 takes 48 seconds on an AMD-64 3000+).

2. Using a exponential distribution, the simulation was run twice for 256 random network sizes were between 64 and 16,777,216.

For each simulation the routing table was created for 256 random nodes in the network. The routing table created accurately reflects what the routing table would look like in an idealized DistribNet network. If there were less then 256 nodes in the network than the routing table was created for every node in the network. For each node sampled, various statistics were gathered. The primary ones of interest are the size of the routing table and the size of the leaf set. Figure 6.1 shows the size of the routing table for each network size simulated while Figure 6.2 shows the size of the leaf set. For clarity only the first simulation for each network size was shown. The same graphs, but with the results of the second simulation, the complete data set, with the statistics for every run, and the source code for the simulation can be found in Appendix A.

As shown in Figure 6.1 the size of the routing table is roughly logarithmic to the network size, however, there is a large variance in the size of the routing table between nodes. Furthermore, certain network sizes lead to larger than average routing table sizes. A similar pattern can be found in the sizes of the leaf set as show in Figure 6.2. The size of the leaf set does not grow with the network size, however there are certain values where the maximum leaf set size can get very large. These spikes are around 2^{11}, 2^{15}, 2^{19}, 2^{23}. Each of these numbers are half of even powers of 16: 2^{12}, 2^{16}, 2^{20}, 2^{24}. After these spikes, the maximum leaf set size drops sharply. Another interesting thing to note is that while the variance in the leaf set size peaks at half of the even powers of 16 the average leaf set size peaks at a fourth of the even powers of 16: 2^{10}, 2^{14}, 2^{18}, 2^{22}, after which the average leaf set size drops while the maximum leaf set size continues to increase. In fact even when the maximum leaf set size drops the average leaf set size continues to grow. I am, unfortunately, not exactly sure as to the cause of these spikes. I am sure, however, that these results accurately reflect what will happen in the real DistribNet network for several reasons:

1. The generated routing tables are identical to what they will be in a stable DistribNet network with the same nodes.

2. The results are peculiar but not random. For each spike in variance there is a gradual buildup and then a sharp drop.

3. I ran each simulation at least twice, and the results are approximately the same.

The large variance in the routing table sizes may be cause for concern since each node in the routing table needs to be maintained by periodically sending packets to the node. Whether this overhead is significant needs to be studied further.

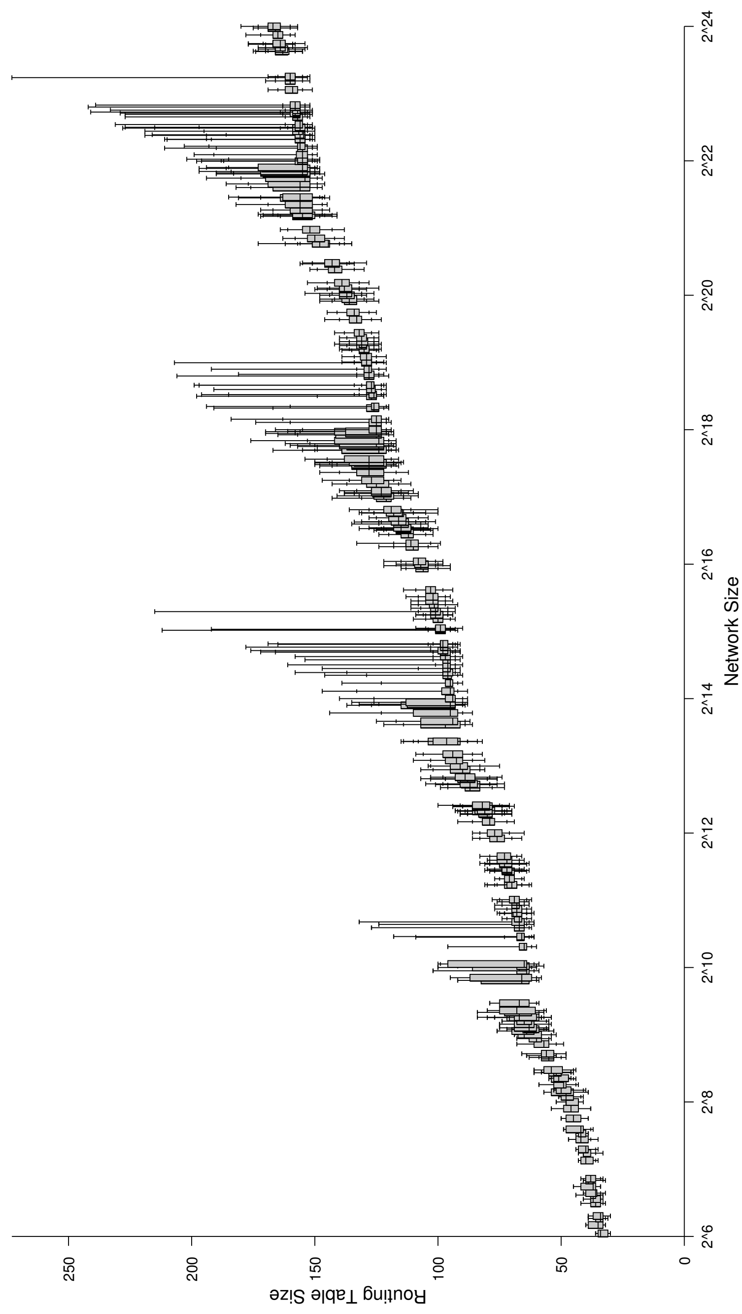

Figure 6.1: The effect of the network size on the routing table size. The box plots show the 25/Medium/75% where the lines stretch to the minimum and maximum with (barely visible) ticks at the 5/95%.

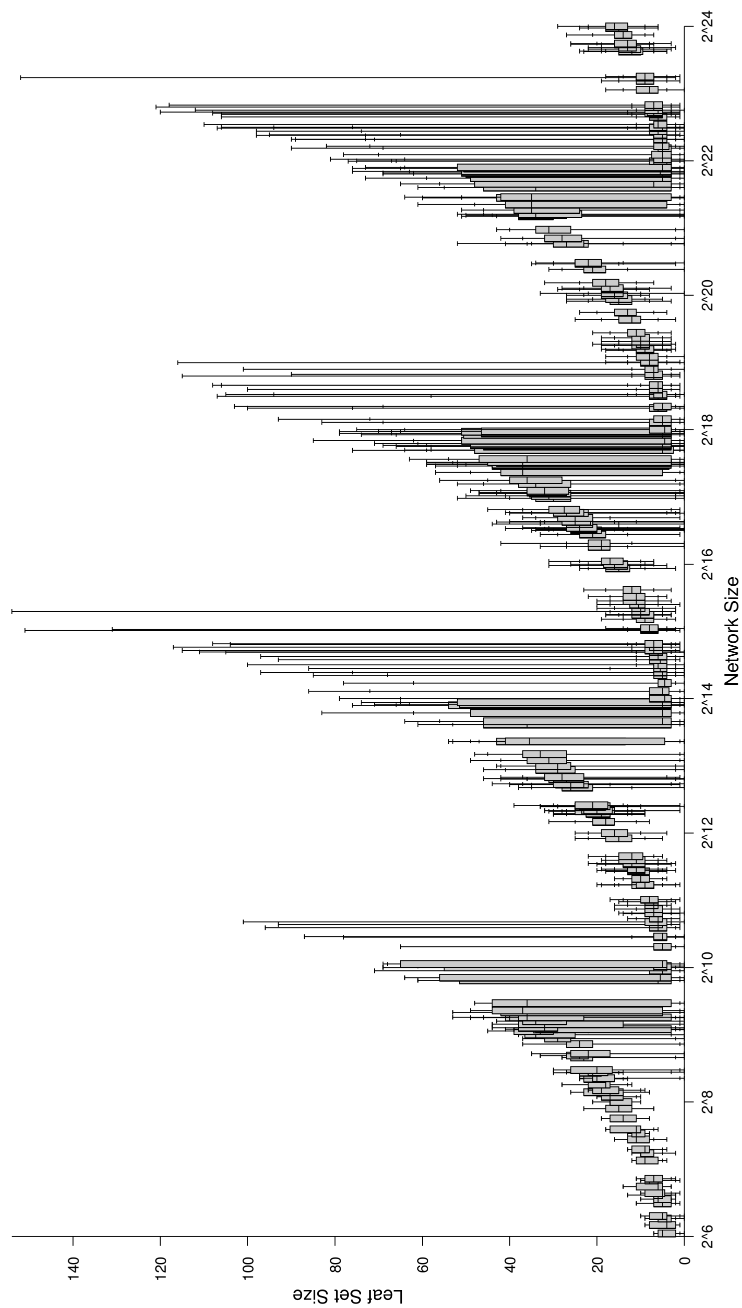

Figure 6.2: The effect of the network size on the leaf set size. The box plots show the 25/Medium/75% where the lines stretch to the minimum and maximum with (barely visible) ticks at the 5/95%.

Chapter 7

Limitations and Future Directions

The current design of DistribNet is very limited. In this chapter I will discuss these limitations and possible ways to solve them.

7.1 Map Keys and Mutable Documents

DistribNet, as currently implemented, is very limited in the type of data it can store. In particular only non-mutable keys indexed by the hash of their content are supported.

Future versions of DistribNet will support map keys which will be indexed based on their title and can be updated. Map keys will contain the following information:

- Short Description
- Public Namespace Key
- Timestamped Index pointers
- Timestamped Data pointers

At any given point in time each map key will only be associated with one index pointer and one data pointer. Map keys can be updated by appending a new index or data pointer to the existing list. By default, when a map key is queried only the most recent pointer will be returned. However, older pointers are still there and may be retrieved by specifying a specific date. Thus, map keys may be updated, but information is never lost or overwritten.

7.2 Freeing Space and Determining Worth

One important aspect of DistribNet is to determine how valuable a key is and thus how hard the network should try to keep it available. Worth should not be based on how popular a file is at the moment but how popular a file is over the long term. Furthermore a file which is popular over a long period of time should be given more value than a file which is really popular but only for a short period of time. Worth should be based on a combination of long term access patterns as well as what users think of the file. Authors should also be able to influence the worth of a document, for example a draft of a paper should have less worth than the published paper.

In order to determine worth based on usage patterns access to a file must be logged in some way. Unfortunately logging every single access of a file from the beginning of time is not very practical so it needs to be summarized in some manner that will accurately reflect overall usage patterns. Simply maintaining a count of the number of times a file is accessed is not sufficient as that will not be able to distinguish files which are popular over a long period over file which are really popular but only for a short time. It also needs to be possible to merge the access logs of different nodes or at least combine their worth rankings in some intelligent manner.

In order to determine worth based on users' opinion of a file there needs to be a way for users to vote on a file. Furthermore, there needs to be a way to prevent a user from voting multiple times and thus artificially increasing the worth of a file.

Authors should also have a say on the worth of a file. However, they should only be able to influence so much. Naturally they should not be able to give a file a high worth value, but they should also not be able to give it too low of a value. For example an author should not be able to release a document which becomes very popular and then some time latter deem it is worthless and thus remove it from the network. In DistribNet I plan to allow a author to mark a document as deleted, but won't allow it to be actually purged. Rather, deleting a document will lower its worth value by a small percentage. If no one was interested in the file in the first place then this small decrease in worth will likely cause the file to "fall off" the network fairly quickly. However if lots of people are still accessing a file it will still be available.

Thus, determining worth of a document is far from simple. Because of this the current version of DistribNet makes no attempt to determine worth and thus no attempt to remove worthless documents. This means that eventually the DistribNet network will run out of space. If this happens the only way to resolve the situation is to manually remove files from the store. This is obviously a serious limitation and needs to be addressed somehow before DistribNet can be used as a general purpose file sharing network available to the public.

7.3 Anonymity

Because there is no indirection when retrieving data, most of the data on any particular node would be data that a local node user requested at some point in time. This means that it is fairly easy to tell which keys a particular user requested. Although complete anonymity for the browser is not one of my goals this is going a bit too far. One solution for this is to do something similar to what GNUNet does, which is described in [6].

It is also blatantly obvious which nodes have which keys. Although I do not see this as a major problem, especially if a solution for the first problem is found, it is something to consider.

7.4 Using BitTorrent For Content Distribution

The distribution of Data keys in DistribNet is rather simplistic. They are simply stored where they are used last. However, since the distribution of data keys is independent from the indexing of these keys a completely different system can be used.

One such system is BitTorrent [2] which was first introduced shortly after DistribNet was started. In this very short time BitTorrent has become extremely popular due to its efficient way of distributing large files. BitTorrent's strategy of distributing files is similar to the way Data keys are distributed in DistribNet but more advanced. BitTorrent works by requiring everyone who is currently downloading a file to also upload the file to other downloaders. A file is not downloaded in sequential order, instead random pieces are downloaded. Which pieces depends on which pieces are available by other users trying to download the same file. Eventually the complete file will be download as each downloader will have a different part of the file available to share.

To distribute a file on BitTorrent:

1. A `.torrent` file which bootstraps the process needs to be made available via conventional means (often a web site). Since the this file is generally small, it can easily be distributed as an index key with DistribNet.

2. A complete copy of the file needs to be made available. In DistribNet's case, the file

can be hosted on the DistribNet network itself as a permanent data key.

3. Finally, a *tracker* needs to be made available. A *tracker* is a server that allows downloaders to find each other. If the tracker goes down than no one else can download the file. Thus the tracker in the one non-distributed aspect of BitTorrent. To implement a tracker in DistribNet a random node can be selected. Because the downloading of files is distributed the tracker itself will receive very little traffic. Unfortunately, if the node goes down than the file will be lost. A possible solution to this problem is to have multiple trackers for a particular file. If one of the tracker's goes down all a downloader needs to do is switch to another one. For any one file actively being downloaded several trackers will be maintained. If a file is not being download than no trackers will be maintained, however once someone starts downloading the file 2 or 3 random nodes will be selected to serve as trackers.

Since, BitTorrent solves the problem of distributing the network load very well, the use of the all or part of the BitTorrent protocol, instead of the exiting, rather simplistic, protocol in use now by DistribNet, is definitely worth investigating.

Chapter 8

Conclusion

In this paper I have presented a distributed file sharing network known as DistribNet. The goal of designing DistribNet was to prove that it is possible to develop a system which anyone can tap into or add content to and that focuses on the long term availability of documents.

In order to build a system which addresses this problem I had to implement some sort of DHT, which in itself turned out to be a rather interesting problem and is the main contribution of this Thesis. The primary goal of a DHT is to be able to route messages to the appropriate node. DistribNet unique routing strategy is a combination of Pastry's routing table and Kademlia XOR based metric. It is completely decentralized, scalable, and self-organizing as it automatically adopts to the arrival, departure and failure of nodes like Pastry does. Yet its routing strategy is simpler than Pastry thanks to the XOR based metric as found in Kademlia.

The routing table in DistribNet is used to find the location of the content. The actual storage and retrieval of data is independent of the routing. DistribNet uses a simpler and less developed system for this part. However, being independent of the routing, a completely different strategy can be used instead, such as one similar to what BitTorrent uses.

DistribNet addresses the issue of the long term availability of documents by never removing documents from the system. However, this strategy will not scale well over time. The issue of when to remove documents from the system, while still maintaining the availability of ones which may be of use to some one in the future, is a difficult problem which needs to be studied further.

Appendix A

Routing Table Simulation

This appendix gives the graphs of the second run and the complete data set from all the runs in the routing table simulation described in Chapter 6.2. The source code to the simulation is also provided.

A.1 Graphs of the Second Run

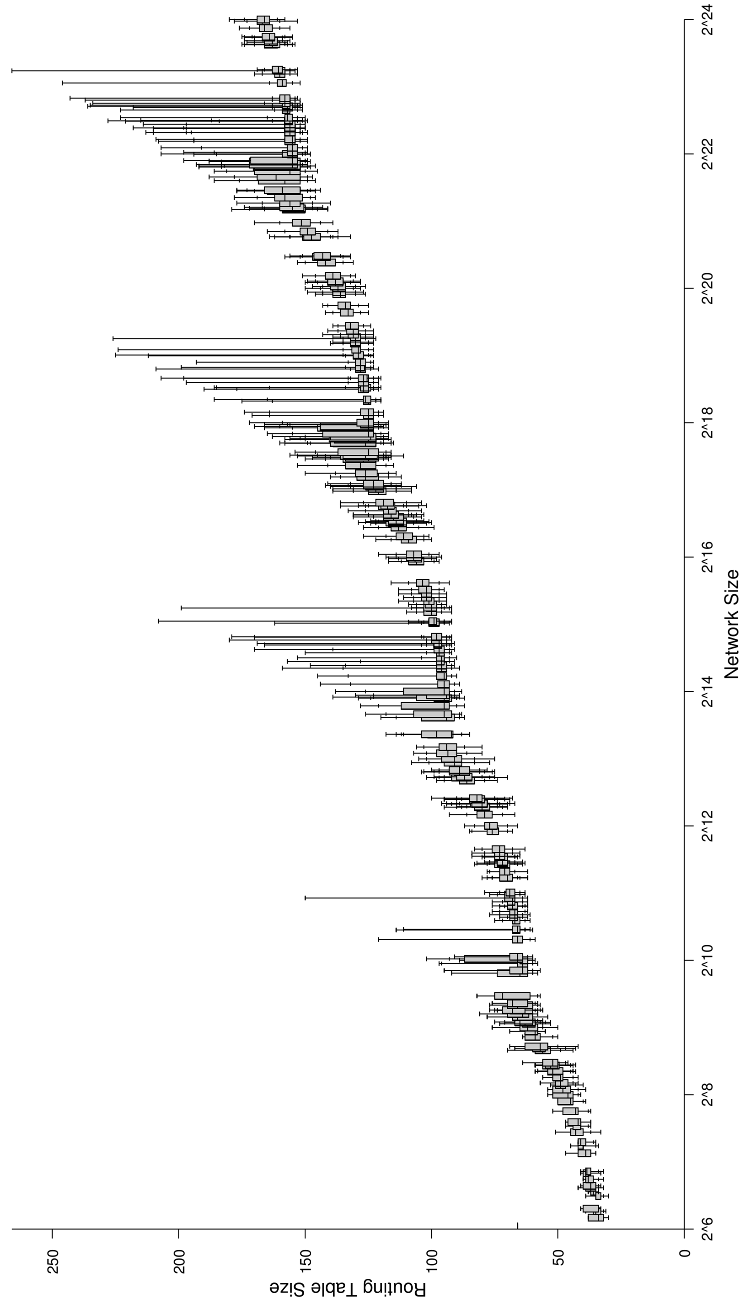

Figure A.1: The effect of the network size on the routing table size. This plot is identical to Figure 6.1 except that the data is taken from the second run instead of the first. The box plots show the 25/Medium/75% where the lines stretch to the minimum and maximum with (barely visible) ticks at the 5/95%.

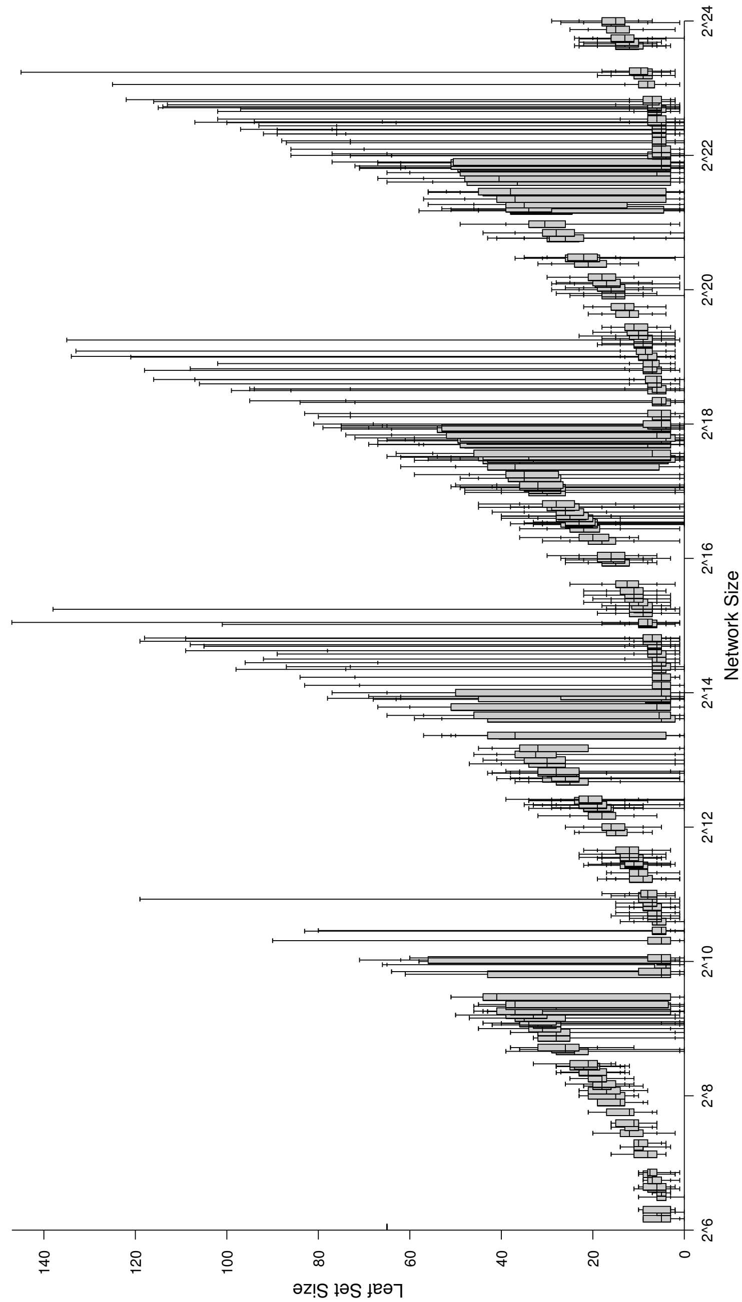

Figure A.2: The effect of the network size on the leaf set size. This plot is identical to Figure 6.2 except that the data is taken from the second run instead of the first. The box plots show the 25/Medium/75% where the lines stretch to the minimum and maximum with (barely visible) ticks at the 5/95%.

A.2 Complete Data Set

The columns of both charts are as follows:

- Average
- Standard Deviation

- Mean
- Difference between the 75th and 25th percentile
- Difference between the 95th and 5th percentile
- Difference between the maximum and the minimum

- Minimum
- 5th percentile
- 25th percentile
- Mean
- 75th percentile
- 95th percentile
- Maximum

A.2.1 Routing Table Size Data

66	33	2.1	34	4	6	6	30	30	31	34	35	36	36
66	66	0.0	66	0	0	0	66	66	66	66	66	66	66
72	36	2.7	35	6	8	8	32	32	33	35	39	40	40
72	34	2.8	34	6	8	8	30	30	32	34	38	38	38
77	35	2.3	34	3	7	8	31	32	33	34	36	39	39
77	35	2.1	35	3	7	8	31	32	33	35	36	39	39
79	35	2.6	34	4	9	9	30	30	33	34	37	39	39
79	36	2.9	34	6	8	8	33	33	34	34	40	41	41
90	37	2.7	36	4	9	10	32	33	34	36	38	42	42

90	35	2.4	35	2	7	9	30	32	33	35	35	39	39
94	37	2.3	37	4	7	8	33	34	34	37	38	41	41
94	36	1.3	36	2	4	4	34	34	35	36	37	38	38
98	38	3.3	37	2	11	11	33	33	36	37	38	44	44
98	37	2.6	37	4	8	10	32	34	35	37	39	42	42
100	37	2.4	36	4	7	9	32	34	36	36	40	41	41
100	37	2.7	37	5	7	8	33	34	35	37	40	41	41
107	38	3.6	37	6	11	11	34	34	36	37	42	45	45
107	37	1.8	38	3	6	8	32	34	36	38	39	40	40
114	38	2.4	39	3	8	9	32	33	37	39	40	41	41
114	38	2.5	39	2	8	8	33	33	38	39	40	41	41
116	38	2.2	38	4	8	9	33	34	36	38	40	42	42
116	38	2.1	39	2	7	9	32	34	37	39	39	41	41
140	40	2.7	40	5	7	8	35	36	37	40	42	43	43
140	40	3.5	39	5	12	12	35	35	37	39	42	47	47
151	40	2.1	41	3	7	10	33	36	38	41	41	43	43
151	40	2.7	40	2	10	11	34	35	40	40	42	45	45
157	41	2.6	40	4	8	9	35	36	39	40	43	44	44
157	40	1.8	41	3	6	7	35	36	39	41	42	42	42
174	42	2.8	42	5	9	12	35	38	39	42	44	47	47
174	43	4.5	43	5	14	18	33	37	40	43	45	51	51
185	42	1.6	42	3	5	5	39	39	40	42	43	44	44
185	42	2.6	41	3	9	10	37	38	41	41	44	47	47
193	43	3.7	42	7	11	12	37	38	41	42	48	49	49
193	43	3.0	42	5	10	10	37	37	41	42	46	47	47
216	44	3.5	45	6	11	11	39	39	42	45	48	50	50
216	45	3.9	43	6	14	15	37	38	42	43	48	52	52
239	46	4.1	46	6	16	16	38	38	43	46	49	54	54
239	46	3.4	45	6	10	11	39	40	44	45	50	50	50
239	46	4.0	46	6	13	14	40	41	43	46	49	54	54
239	46	3.7	46	6	13	13	39	39	43	46	49	52	52
256	47	3.3	48	5	9	11	41	43	43	48	48	52	52
256	47	4.0	46	8	12	13	41	42	44	46	52	54	54
256	47	3.2	47	3	12	13	39	40	45	47	48	52	52
256	47	3.5	48	5	12	13	39	40	44	48	49	52	52
256	47	4.5	47	7	17	17	38	38	44	47	51	55	55
270	47	2.8	48	5	9	10	41	42	45	48	50	51	51
270	48	3.9	48	7	12	15	39	42	45	48	52	54	54
283	49	5.3	48	9	18	18	39	39	45	48	54	57	57
283	48	3.3	49	4	10	13	40	43	47	49	51	53	53
289	49	3.8	50	6	12	13	40	41	46	50	52	53	53
289	49	3.9	49	5	13	15	42	44	46	49	51	57	57
306	50	4.3	49	5	15	16	43	44	48	49	53	59	59
306	50	3.3	49	4	12	14	42	44	48	49	52	56	56
325	51	3.4	52	5	10	11	44	45	49	52	54	55	55
325	51	4.1	50	5	14	16	43	45	49	50	54	59	59
327	51	2.9	51	6	8	9	46	47	47	51	53	55	55
327	51	4.0	52	6	14	14	44	44	48	52	54	58	58
346	53	4.4	53	5	15	16	45	46	50	53	55	61	61
346	52	4.2	53	6	14	15	44	45	50	53	56	59	59
348	52	3.3	52	3	12	12	46	46	50	52	53	58	58
348	53	4.1	53	5	14	16	43	45	50	53	55	59	59
356	53	4.8	54	7	16	17	44	45	50	54	57	61	61
356	53	4.8	52	6	17	18	46	47	50	52	56	64	64

405	55	3.3	55	5	9	15	48	50	53	55	58	59	63
405	57	5.8	56	6	21	26	44	49	53	56	59	70	70
413	56	3.8	55	5	12	12	52	52	53	55	58	64	64
413	57	5.1	57	5	20	24	43	47	55	57	60	67	67
421	56	4.2	56	5	18	18	48	48	53	56	58	66	66
421	58	5.9	57	9	19	27	42	50	54	57	63	69	69
465	58	4.9	57	6	16	19	49	52	55	57	61	68	68
465	59	3.9	59	6	12	14	50	52	57	59	63	64	64
492	61	3.8	60	5	13	14	54	55	58	60	63	68	68
492	61	4.3	59	5	14	14	55	55	58	59	63	69	69
512	63	5.3	65	10	15	17	52	54	58	65	68	69	69
512	62	5.9	62	7	20	26	50	56	58	62	65	76	76
532	65	6.6	64	9	23	23	53	53	61	64	70	76	76
532	62	4.8	62	5	17	20	53	56	59	62	64	73	73
540	63	3.7	63	7	13	15	55	57	60	63	67	70	70
540	63	4.8	62	8	16	18	53	55	59	62	67	71	71
542	63	4.6	64	6	16	17	55	56	59	64	65	72	72
542	63	4.8	63	5	18	19	56	57	60	63	65	75	75
550	64	5.3	63	9	19	20	55	56	60	63	69	75	75
550	63	3.5	65	7	9	11	56	58	60	65	67	67	67
571	64	5.8	63	8	20	21	54	55	61	63	69	75	75
571	65	6.3	66	8	24	24	54	54	60	66	68	78	78
589	65	5.1	65	6	18	19	55	56	62	65	68	74	74
589	66	6.3	64	8	23	23	58	58	62	64	70	81	81
611	67	6.6	66	11	22	22	58	58	61	66	72	80	80
611	65	5.3	65	8	15	16	58	59	61	65	69	74	74
612	65	6.6	65	12	20	23	54	57	59	65	71	77	77
612	65	6.1	64	10	19	19	56	56	60	64	70	75	75
613	66	7.4	67	9	26	30	54	58	60	67	69	84	84
613	68	5.7	68	9	20	21	56	57	63	68	72	77	77
645	68	7.2	67	11	25	27	57	59	62	67	73	84	84
645	66	5.8	66	10	19	20	57	58	60	66	70	77	77
657	68	7.4	68	14	23	24	56	57	61	68	75	80	80
657	67	4.8	68	8	17	18	58	59	62	68	70	76	76
709	68	6.1	67	12	19	20	59	60	63	67	75	79	79
709	68	7.4	72	14	24	25	57	58	61	72	75	82	82
897	72	10.8	66	21	32	33	59	60	63	66	84	92	92
897	68	9.1	65	12	32	34	58	60	62	65	74	92	92
922	74	13.2	66	25	35	37	58	60	62	66	87	95	95
922	69	11.7	64	7	35	38	57	60	62	64	69	95	95
990	70	12.3	64	5	41	43	59	61	63	64	68	102	102
990	68	11.6	64	4	36	39	58	60	62	64	66	96	97
1024	74	12.7	67	22	38	40	60	62	64	67	86	100	100
1024	66	7.0	64	3	29	30	59	60	62	64	65	89	89
1024	72	12.4	65	18	36	38	59	61	63	65	81	97	97
1039	68	9.6	64	5	32	35	57	60	63	64	68	92	92
1039	74	13.2	66	23	31	42	60	62	64	66	87	93	102
1062	74	15.1	65	32	38	41	59	61	64	65	96	99	100
1062	70	9.7	66	5	29	31	60	62	64	66	69	91	91
1269	67	7.7	65	3	34	36	60	62	64	65	67	96	96
1269	71	16.4	66	4	60	62	59	61	64	66	68	121	121
1400	68	7.8	66	3	10	48	61	63	65	66	68	73	109
1400	69	12.1	66	3	51	54	60	63	65	66	68	114	114
1409	69	12.4	66	3	56	57	61	62	65	66	68	118	118

1409	69	10.5	66	3	48	50	61	63	65	66	68	111	111
1544	72	17.5	67	4	64	65	62	63	65	67	69	127	127
1544	67	2.8	67	3	9	14	61	63	65	67	68	72	75
1597	70	13.6	67	4	60	63	61	64	65	67	69	124	124
1597	67	2.3	67	3	6	11	62	64	66	67	69	70	73
1638	71	16.4	67	5	69	71	61	63	65	67	70	132	132
1638	67	2.4	67	3	9	16	61	64	66	67	69	73	77
1695	68	2.5	67	3	8	12	62	64	66	67	69	72	74
1695	68	2.9	67	3	10	14	62	63	66	67	69	73	76
1787	68	2.9	68	4	9	13	62	64	66	68	70	73	75
1787	68	2.4	68	4	7	11	62	64	66	68	70	71	73
1795	68	2.8	68	4	9	15	61	64	66	68	70	73	76
1795	68	3.0	68	4	10	14	62	63	66	68	70	73	76
1871	68	2.6	68	4	8	15	62	64	66	68	70	72	77
1871	68	2.5	68	3	8	14	62	64	67	68	70	72	76
1947	68	2.7	68	3	9	14	63	65	67	68	70	74	77
1947	74	20.5	68	4	86	88	62	64	67	68	71	150	150
2016	68	2.5	68	3	9	13	63	65	67	68	70	74	76
2016	69	2.7	69	4	9	14	63	65	67	69	71	74	77
2061	69	3.1	69	4	11	16	62	64	67	69	71	75	78
2061	69	2.5	69	4	8	16	63	65	67	69	71	73	79
2397	71	3.2	71	5	11	17	63	66	68	71	73	77	80
2397	70	3.3	70	5	11	18	62	65	68	70	73	76	80
2403	70	3.1	70	4	10	19	62	66	68	70	72	76	81
2403	70	3.1	70	5	10	16	62	66	68	70	73	76	78
2555	71	2.7	71	4	9	12	65	66	69	71	73	75	77
2555	71	2.9	71	4	10	16	62	67	69	71	73	77	78
2760	72	3.1	72	5	10	15	64	67	69	72	74	77	79
2760	72	3.7	71	6	12	19	64	66	69	71	75	78	83
2790	72	3.4	72	4	11	18	63	66	70	72	74	77	81
2790	72	3.3	72	4	10	19	63	67	70	72	74	77	82
2825	72	3.1	72	4	10	15	65	67	70	72	74	77	80
2825	72	2.8	72	4	10	15	64	67	70	72	74	77	79
2971	73	3.4	73	5	12	17	64	67	70	73	75	79	81
2971	73	3.3	72	5	12	14	66	67	70	72	75	79	80
2996	73	3.8	73	5	12	20	63	67	70	73	75	79	83
2996	73	3.5	73	4	12	19	65	67	71	73	75	79	84
3096	73	3.6	72	6	12	15	65	67	70	72	76	79	80
3096	73	3.6	73	4	11	19	65	68	71	73	75	79	84
3224	73	3.4	73	5	11	17	66	68	71	73	76	79	83
3224	73	3.5	73	5	12	20	63	68	71	73	76	80	83
3878	76	4.1	76	6	13	20	66	70	73	76	79	83	86
3878	76	3.7	76	4	14	17	68	70	74	76	78	84	85
4096	77	3.9	77	6	12	21	65	71	74	77	80	83	86
4096	77	3.9	77	5	13	21	66	70	74	77	79	83	87
4096	77	3.8	77	6	11	21	67	72	74	77	80	83	88
4602	79	4.1	79	5	14	23	69	72	77	79	82	86	92
4602	79	4.4	79	6	14	26	67	72	76	79	82	86	93
4967	81	4.2	81	5	14	21	70	74	78	81	83	88	91
4967	80	4.8	79	6	16	25	70	72	77	79	83	88	95
5000	81	4.5	80	6	15	21	70	73	78	80	84	88	91
5000	80	3.9	80	6	12	20	70	74	77	80	83	86	90
5130	82	4.7	82	7	16	22	70	73	79	82	86	89	92
5130	81	5.0	81	6	19	24	70	73	78	81	84	92	94

5156	81	4.5	81	8	14	21	72	74	77	81	85	88	93
5156	81	4.9	81	7	15	27	67	74	78	81	85	89	94
5161	81	4.9	81	6	17	23	70	74	78	81	84	91	93
5161	81	4.6	80	6	16	27	69	74	78	80	84	90	96
5375	82	5.0	82	7	16	23	71	74	79	82	86	90	94
5375	82	4.7	82	6	16	26	69	74	79	82	85	90	95
5402	82	4.6	82	7	16	25	69	74	78	82	85	90	94
5402	82	4.4	82	5	13	24	71	75	79	82	84	88	95
5454	82	4.7	82	7	16	29	71	75	79	82	86	91	100
5454	82	4.6	82	5	15	32	68	75	80	82	85	90	100
6545	87	5.1	87	6	18	26	73	78	83	87	89	96	99
6545	86	4.9	86	6	16	24	74	79	83	86	89	95	98
6752	87	5.1	88	8	17	25	76	79	83	88	91	96	101
6752	88	5.6	88	8	20	32	70	77	84	88	92	97	102
6800	88	5.4	87	7	19	32	73	79	84	87	91	98	105
6800	87	4.8	87	6	15	24	75	80	84	87	90	95	99
7144	88	5.4	88	7	17	27	76	80	85	88	92	97	103
7144	88	4.9	88	6	16	28	76	81	85	88	91	97	104
7173	88	4.9	88	6	17	31	76	80	85	88	91	97	107
7173	89	5.8	89	8	20	28	75	79	85	89	93	99	103
7290	89	5.7	89	8	19	29	74	79	85	89	93	98	103
7290	89	4.8	89	8	16	22	78	81	85	89	93	97	100
7866	91	5.6	90	8	19	26	81	83	87	90	95	102	107
7866	92	5.4	91	7	18	31	77	83	88	91	95	101	108
8175	91	5.4	91	7	20	29	75	83	88	91	95	103	104
8175	92	5.6	91	8	19	30	75	83	88	91	96	102	105
8668	93	5.3	93	7	17	29	81	86	90	93	97	103	110
8668	94	5.3	94	8	16	27	80	86	90	94	98	102	107
9238	94	5.6	94	8	20	27	82	86	90	94	98	106	109
9238	94	4.9	94	7	16	26	80	87	90	94	97	103	106
10525	98	6.9	100	13	20	33	82	88	91	100	104	108	115
10525	97	7.0	97	10	23	29	85	88	92	97	102	111	114
10540	97	6.8	97	10	22	30	84	88	92	97	102	110	114
10540	98	7.6	98	12	24	33	85	88	92	98	104	112	118
12535	100	9.2	97	16	25	36	86	89	91	97	107	114	122
12535	98	8.5	94	13	25	33	87	89	91	94	104	114	120
12969	99	9.4	94	15	28	38	87	89	92	94	107	117	125
12969	99	9.8	95	15	29	38	88	89	92	95	107	118	126
14133	101	11.2	95	18	33	58	86	90	92	95	110	123	144
14133	102	11.2	95	19	31	41	87	90	93	95	112	121	128
15298	102	13.3	95	22	37	48	89	90	93	95	115	127	137
15298	100	11.6	94	7	34	42	87	90	92	94	99	124	129
15464	101	11.9	95	20	34	44	88	90	93	95	113	124	132
15464	100	11.4	95	13	32	50	89	91	93	95	106	123	139
15768	102	12.9	95	20	36	47	88	90	93	95	113	126	135
15768	100	11.3	94	11	32	41	89	91	93	94	104	123	130
16384	100	12.3	94	4	36	52	88	90	93	94	97	126	140
16384	101	12.6	95	18	35	50	88	91	93	95	111	126	138
16384	101	13.1	95	16	37	51	90	91	93	95	109	128	141
17684	102	14.0	95	5	41	59	88	92	94	95	99	133	147
17684	100	12.5	95	5	41	55	89	91	93	95	98	132	144
19226	97	8.9	95	3	31	49	90	92	94	95	97	123	139
19226	100	13.3	95	4	41	55	90	92	94	95	98	133	145
20849	99	10.0	96	3	37	56	90	92	95	96	98	129	146

20849	99	12.0	96	4	43	70	89	92	94	96	98	135	159
21442	99	11.7	96	4	44	67	91	93	94	96	98	137	158
21442	99	11.2	96	4	42	57	91	92	94	96	98	134	148
22342	98	9.9	96	3	15	56	91	93	95	96	98	108	147
22342	99	10.7	96	3	35	66	91	93	95	96	98	128	157
23202	98	9.8	96	3	8	71	90	93	95	96	98	101	161
23202	98	9.9	96	3	11	63	90	93	95	96	98	104	153
24436	98	10.5	96	3	9	63	91	93	95	96	98	102	154
24436	98	8.0	97	3	9	58	92	93	95	97	98	102	150
25278	98	8.0	97	4	9	68	90	93	95	97	99	102	158
25278	100	13.4	97	4	46	79	91	93	95	97	99	139	170
26466	98	7.2	98	3	8	75	91	94	96	98	99	102	166
26466	99	11.2	97	3	10	74	92	94	96	97	99	104	166
26496	98	7.0	97	4	9	81	91	93	95	97	99	102	172
26496	98	8.1	97	3	9	74	92	93	96	97	99	102	166
26894	100	11.9	98	4	10	85	91	93	96	98	100	103	176
26894	98	8.6	97	3	8	78	91	94	96	97	99	102	169
27865	98	7.5	98	3	9	86	92	94	96	98	99	103	178
27865	98	8.0	97	4	8	88	92	94	96	97	100	102	180
28676	99	7.3	98	3	11	74	91	93	96	98	99	104	165
28676	98	6.0	98	4	10	78	92	93	96	98	100	103	170
28823	99	7.3	98	3	8	77	92	94	96	98	99	102	169
28823	99	8.4	98	4	10	87	92	94	96	98	100	104	179
33146	101	14.7	99	4	9	120	92	95	97	99	101	104	212
33146	99	4.9	99	4	9	69	93	95	97	99	101	104	162
33494	99	6.5	99	4	9	99	93	95	97	99	101	104	192
33494	99	2.8	99	4	8	17	92	95	97	99	101	103	109
33855	99	3.0	99	4	10	19	90	95	97	99	101	105	109
33855	100	10.0	99	3	10	116	92	95	98	99	101	105	208
37192	100	3.1	100	4	10	17	93	95	98	100	102	105	110
37192	101	3.1	100	5	10	18	92	96	98	100	103	106	110
38763	100	2.7	100	4	9	15	94	96	98	100	102	105	109
38763	101	6.9	100	4	11	106	93	95	98	100	102	106	199
38908	101	3.1	101	5	10	15	94	96	98	101	103	106	109
38908	101	3.1	100	5	10	16	92	96	98	100	103	106	108
40188	102	10.5	101	4	12	122	93	96	99	101	103	108	215
40188	101	3.0	100	5	10	15	94	96	98	100	103	106	109
41640	102	3.1	101	3	11	18	93	96	100	101	103	107	111
41640	101	3.2	101	4	10	19	94	97	99	101	103	107	113
43155	102	3.2	102	4	11	19	92	97	100	102	104	108	111
43155	102	3.2	102	4	10	17	94	97	100	102	104	107	111
44845	102	3.4	102	5	11	17	94	97	100	102	105	108	111
44845	102	3.3	102	4	11	19	94	97	100	102	104	108	113
46825	103	3.4	102	5	11	18	95	97	100	102	105	108	113
46825	102	3.4	102	5	11	18	95	97	100	102	105	108	113
50226	103	3.4	103	4	11	20	94	98	101	103	105	109	114
50226	103	3.9	104	5	12	23	93	97	101	104	106	109	116
62700	106	4.0	106	5	13	20	95	100	104	106	109	113	115
62700	106	3.9	106	6	13	20	97	99	103	106	109	112	117
64497	107	4.4	107	5	14	27	95	101	104	107	109	115	122
64497	107	3.8	107	5	13	19	98	100	104	107	109	113	117
65536	107	3.8	107	6	13	19	98	100	104	107	110	113	117
65536	107	4.0	107	6	14	22	96	100	104	107	110	114	118
65536	107	4.0	107	6	13	23	98	101	104	107	110	114	121

67433	108	4.0	108	5	14	24	98	101	105	108	110	115	122
67433	107	4.1	107	6	13	24	97	101	104	107	110	114	121
78442	110	4.1	110	5	14	24	100	104	108	110	113	118	124
78442	109	4.3	109	6	13	22	100	103	106	109	112	116	122
81254	111	4.6	111	5	15	34	99	103	108	111	113	118	133
81254	111	4.6	111	6	15	26	101	103	108	111	114	118	127
88955	113	4.4	112	5	15	22	102	105	110	112	115	120	124
88955	113	4.8	113	6	16	28	99	105	110	113	116	121	127
92905	114	4.4	114	7	16	24	102	106	110	114	117	122	126
92905	114	4.5	114	7	15	22	102	107	110	114	117	122	124
93623	114	4.2	114	6	14	22	103	107	111	114	117	121	125
93623	114	4.6	114	7	15	29	100	107	111	114	118	122	129
94561	114	4.9	114	6	17	32	100	106	111	114	117	123	132
94561	114	4.6	114	7	15	25	101	106	110	114	117	121	126
95372	115	4.7	115	7	15	24	104	107	111	115	118	122	128
95372	114	4.7	114	6	16	21	103	106	111	114	117	122	124
99160	115	5.0	115	6	17	31	104	107	112	115	118	124	135
99160	115	5.1	115	8	16	27	104	107	111	115	119	123	131
101225	116	4.5	115	6	14	29	105	109	112	115	118	123	134
101225	116	4.7	116	6	16	25	106	109	113	116	119	125	131
101559	116	4.7	116	6	15	30	101	109	113	116	119	124	131
101559	116	4.9	116	6	17	28	103	108	113	116	119	125	131
105691	116	4.9	116	7	17	24	104	108	113	116	120	125	128
105691	117	4.9	117	5	17	29	104	109	114	117	119	126	133
110967	118	4.9	117	6	16	31	100	110	114	117	120	126	131
110967	118	4.8	118	6	16	31	105	110	115	118	121	126	136
111404	118	5.1	118	6	16	27	105	110	115	118	121	126	132
111404	118	4.5	118	5	14	27	102	111	115	118	120	125	129
114879	119	5.2	119	7	17	36	100	111	115	119	122	128	136
114879	119	5.2	119	7	17	32	104	110	115	119	122	127	136
129414	122	5.2	121	6	17	32	111	114	119	121	125	131	143
129414	122	5.2	121	7	17	31	108	114	118	121	125	131	139
132549	122	5.8	122	8	18	33	108	114	118	122	126	132	141
132549	122	5.2	122	6	17	31	108	114	119	122	125	131	139
135777	123	5.7	122	8	19	30	108	115	119	122	127	134	138
135777	122	5.6	122	7	18	34	106	114	119	122	126	132	140
137216	123	5.3	123	8	18	26	112	116	119	123	127	134	138
137216	123	5.5	123	8	17	30	112	116	119	123	127	133	142
139718	123	5.3	123	8	18	30	110	115	119	123	127	133	140
139718	123	5.3	123	8	16	29	112	116	119	123	127	132	141
149844	125	6.4	125	9	21	32	111	116	120	125	129	137	143
149844	126	5.6	126	9	19	28	112	117	121	126	130	136	140
155579	126	5.9	127	9	18	32	115	118	122	127	131	136	147
155579	127	6.4	126	8	21	36	114	117	122	126	130	138	150
168788	128	6.9	128	11	23	36	112	117	122	128	133	140	148
168788	128	7.6	128	12	23	38	115	118	122	128	134	141	153
180324	129	7.9	128	13	24	32	116	119	122	128	135	143	148
180324	128	7.5	127	13	23	33	117	119	122	127	135	142	150
182842	128	7.7	128	14	23	33	117	118	121	128	135	141	150
182842	127	7.5	125	13	22	31	116	118	121	125	134	140	147
186034	128	8.2	125	13	26	35	115	118	121	125	134	144	150
186034	129	8.4	129	15	24	42	111	118	121	129	136	142	153
187594	129	8.3	128	14	25	36	114	118	122	128	136	143	150
187594	129	8.5	126	13	26	40	116	119	122	126	135	145	156

193543	130	9.1	128	16	26	38	116	119	122	128	138	145	154
193543	129	9.3	125	16	28	38	116	118	121	125	137	146	154
211998	131	10.5	125	17	30	50	117	120	122	125	139	150	167
211998	130	10.1	125	17	29	42	116	119	122	125	139	148	158
212668	130	10.6	124	18	30	39	116	119	121	124	139	149	155
212668	131	10.3	126	18	29	45	115	120	122	126	140	149	160
220475	129	10.1	124	15	29	40	117	120	122	124	137	149	157
220475	130	10.6	124	17	31	42	116	119	122	124	139	150	158
223090	130	10.8	124	19	30	42	118	120	122	124	141	150	160
223090	131	11.0	125	19	32	39	117	120	122	125	141	152	156
227163	130	10.9	125	18	33	45	117	119	122	125	140	152	162
227163	129	10.6	124	18	31	46	117	119	122	124	140	150	163
234239	131	12.2	124	20	33	59	117	120	122	124	142	153	176
234239	132	11.9	125	20	35	48	117	120	123	125	143	155	165
248714	132	12.8	125	19	37	47	118	120	123	125	142	157	165
248714	132	12.7	125	22	34	49	117	121	123	125	145	155	166
252247	131	12.4	124	15	37	50	120	121	123	124	138	158	170
252247	132	13.2	125	21	40	51	119	120	123	125	144	160	170
256849	132	13.0	125	19	37	52	118	121	123	125	142	158	170
256849	130	11.6	125	5	35	48	118	121	123	125	128	156	166
257794	130	11.6	125	16	35	42	119	121	123	125	139	156	161
257794	130	12.1	125	5	36	49	117	121	123	125	128	157	166
262144	130	11.6	125	5	34	47	119	121	123	125	128	155	166
262144	130	12.8	125	7	38	55	117	121	123	125	130	159	172
262144	129	11.8	124	5	36	53	118	121	122	124	127	157	171
282691	131	13.1	125	5	39	55	119	121	123	125	128	160	174
282691	130	12.6	125	4	43	52	119	121	123	125	127	164	171
292185	130	13.3	125	4	42	64	120	121	123	125	127	163	184
292185	131	14.2	125	5	43	55	119	121	123	125	128	164	174
327699	131	13.9	126	5	45	71	120	122	124	126	129	167	191
327699	128	11.0	125	3	41	55	120	122	124	125	127	163	175
333128	128	10.9	126	3	39	74	120	121	124	126	127	160	194
333128	128	11.3	126	3	43	66	120	122	124	126	127	165	186
370321	129	11.8	127	4	27	76	122	122	125	127	129	149	198
370321	131	14.4	127	4	54	69	121	123	125	127	129	177	190
376186	129	11.6	127	3	11	63	122	123	125	127	128	134	185
376186	129	11.0	127	4	41	64	121	123	124	127	128	164	185
376561	128	10.7	126	3	9	75	121	123	125	126	128	132	196
376561	129	10.7	127	3	11	66	120	123	125	127	128	134	186
396823	129	9.5	127	3	9	70	121	123	126	127	129	132	191
396823	129	10.2	127	4	10	76	121	123	125	127	129	133	197
414618	128	7.7	127	3	8	75	122	123	126	127	129	131	197
414618	128	7.7	127	4	9	77	121	123	125	127	129	132	198
415024	130	12.7	127	3	11	78	121	123	126	127	129	134	199
415024	129	10.8	127	3	10	87	120	123	126	127	129	133	207
456137	129	8.9	128	4	9	86	120	124	126	128	130	133	206
456137	129	10.6	128	4	8	88	121	124	126	128	130	132	209
464041	128	4.3	128	4	9	59	122	124	126	128	130	133	181
464041	129	7.7	128	3	10	76	123	124	127	128	130	134	199
489220	129	4.8	128	3	9	71	121	124	127	128	130	133	192
489220	129	4.9	128	4	9	70	123	124	126	128	130	133	193
522924	129	5.7	129	4	9	86	121	125	127	129	131	134	207
522924	129	5.9	129	4	10	89	123	124	127	129	131	134	212
526827	129	2.9	129	4	9	17	122	125	127	129	131	134	139

526827	130	9.0	129	4	10	102	123	125	127	129	131	135	225
556524	129	2.9	129	5	9	18	121	125	127	129	132	134	139
556524	130	6.6	129	4	10	101	123	125	128	129	132	135	224
594179	130	3.1	130	4	10	15	124	125	128	130	132	135	139
594179	130	3.1	130	4	10	17	123	125	128	130	132	135	140
601537	130	2.9	130	4	10	17	123	126	128	130	132	136	140
601537	130	3.1	130	4	10	16	123	125	128	130	132	135	139
623975	131	3.0	131	4	11	16	124	126	129	131	133	137	140
623975	131	6.7	130	4	9	104	122	126	128	130	132	135	226
633731	131	2.9	131	3	10	19	123	126	129	131	132	136	142
633731	131	3.0	130	4	10	16	123	126	128	130	132	136	139
650258	131	3.1	131	4	10	16	124	126	129	131	133	136	140
650258	131	3.2	131	4	10	20	123	126	129	131	133	136	143
675966	131	3.1	131	4	11	16	124	126	129	131	133	137	140
675966	131	3.2	131	5	11	18	123	126	129	131	134	137	141
711698	132	3.3	132	4	11	18	124	127	130	132	134	138	142
711698	132	3.2	132	5	10	15	124	127	129	132	134	137	139
815935	134	3.8	133	5	13	23	123	127	131	133	136	140	146
815935	133	3.3	133	5	11	17	125	128	131	133	136	139	142
878307	134	3.7	134	5	13	20	125	128	132	134	137	141	145
878307	134	3.5	134	5	12	18	125	129	132	134	137	141	143
986780	136	4.1	136	5	14	24	124	129	133	136	138	143	148
986780	136	3.6	136	5	13	20	126	130	134	136	139	143	146
1008012	136	4.0	136	6	12	22	126	130	133	136	139	142	148
1008012	136	3.9	136	5	13	22	127	130	134	136	139	143	149
1048576	137	3.9	137	6	13	19	129	131	134	137	140	144	148
1048576	137	3.8	137	5	13	22	128	131	134	137	139	144	150
1048576	137	4.1	136	5	14	22	127	130	134	136	139	144	149
1069151	137	3.8	137	5	12	28	126	131	135	137	140	143	154
1069151	137	4.1	137	6	13	21	126	130	134	137	140	143	147
1113665	138	3.9	138	5	13	21	129	132	135	138	140	145	150
1113665	138	4.0	138	6	13	22	128	132	135	138	141	145	150
1128563	138	3.9	138	5	12	25	124	132	135	138	140	144	149
1128563	138	4.3	138	6	15	21	128	131	135	138	141	146	149
1192316	139	4.2	139	6	13	25	128	132	136	139	142	145	153
1192316	139	4.2	139	6	14	21	130	132	136	139	142	146	151
1368818	142	4.4	142	6	15	22	130	134	139	142	145	149	152
1368818	142	4.6	142	7	15	22	131	135	138	142	145	150	153
1449558	143	4.4	143	6	14	22	134	137	140	143	146	151	156
1449558	143	4.9	143	7	17	26	132	135	140	143	147	152	158
1463861	143	4.7	143	6	15	26	129	136	140	143	146	151	155
1463861	143	4.5	143	7	15	24	132	136	140	143	147	151	156
1781937	148	5.0	148	7	17	27	135	140	144	148	151	157	162
1781937	148	5.0	148	7	17	25	137	139	144	148	151	156	162
1785742	148	5.4	148	6	18	38	135	138	145	148	151	156	173
1785742	147	4.8	148	7	16	32	132	140	144	148	151	156	164
1884161	150	4.9	150	7	16	25	138	142	146	150	153	158	163
1884161	149	5.1	149	6	17	28	137	141	146	149	152	158	165
2061614	151	5.4	152	7	18	26	138	143	148	152	155	161	164
2061614	152	5.1	152	7	16	31	139	144	148	152	155	160	170
2366039	155	5.4	155	8	16	31	141	148	151	155	159	164	172
2366039	155	6.5	154	9	21	38	141	145	150	154	159	166	179
2392176	155	5.9	155	9	19	28	143	146	150	155	159	165	171
2392176	155	6.1	154	9	19	31	141	147	150	154	159	166	172

2417864	155	6.2	155	8	20	32	141	147	151	155	159	167	173
2417864	156	6.1	155	9	20	31	143	147	151	155	160	167	174
2525166	156	6.0	156	9	20	28	144	147	151	156	160	167	172
2525166	156	6.4	156	8	20	37	140	147	152	156	160	167	177
2672788	157	7.1	156	11	22	37	145	147	151	156	162	169	182
2672788	158	6.9	158	11	21	32	146	148	151	158	162	169	178
2861570	159	7.8	159	12	25	37	144	147	152	159	164	172	181
2861570	159	7.3	160	13	21	31	146	149	152	160	165	170	177
2884139	158	7.7	156	12	24	39	146	148	151	156	163	172	185
2884139	159	7.8	159	14	25	33	144	148	152	159	166	173	177
3184126	160	9.2	156	15	27	35	147	149	152	156	167	176	182
3184126	160	9.3	158	17	27	40	146	149	152	158	169	176	186
3306465	160	9.7	156	17	28	40	146	149	152	156	169	177	186
3306465	161	9.8	162	17	29	41	147	149	152	162	169	178	188
3508605	160	11.0	154	18	31	47	147	149	152	154	170	180	194
3508605	161	10.5	156	18	31	41	145	150	152	156	170	181	186
3663174	161	11.5	155	19	33	44	146	150	152	155	171	183	190
3663174	160	11.5	154	19	34	44	148	149	152	154	171	183	192
3703674	161	11.5	155	20	33	42	148	150	152	155	172	183	190
3703674	160	11.3	154	18	33	46	146	149	152	154	170	182	192
3758810	162	12.1	155	19	34	49	148	150	153	155	172	184	197
3758810	161	11.2	155	19	32	44	149	151	153	155	172	183	193
3878401	161	12.1	154	20	36	49	148	150	152	154	172	186	197
3878401	161	11.7	155	20	33	40	148	150	152	155	172	183	188
3916543	161	12.3	155	20	35	45	149	150	153	155	173	185	194
3916543	161	12.9	155	19	38	49	149	150	153	155	172	188	198
4176245	160	11.8	154	5	36	49	149	151	153	154	158	187	198
4176245	159	11.1	154	4	34	46	148	151	153	154	157	185	194
4194304	159	12.1	155	4	38	48	148	150	153	155	157	188	196
4194304	160	11.8	155	6	35	59	148	150	153	155	159	185	207
4194304	160	12.0	155	6	35	45	150	151	153	155	159	186	195
4263816	159	11.2	155	4	34	54	148	151	153	155	157	185	202
4263816	159	11.2	155	4	36	49	149	150	153	155	157	186	198
4467515	160	12.9	155	5	40	50	149	151	153	155	158	191	199
4467515	161	13.3	155	4	40	58	149	151	153	155	157	191	207
4777900	159	11.7	155	4	39	62	149	151	153	155	157	190	211
4777900	161	13.5	155	4	42	59	149	152	154	155	158	194	208
4874035	159	11.1	156	3	42	54	149	151	154	156	157	193	203
4874035	160	13.5	155	4	42	60	149	152	154	155	158	194	209
5226162	159	11.9	156	4	40	60	150	152	154	156	158	192	210
5226162	160	12.7	156	4	45	59	151	152	154	156	158	197	210
5235935	159	11.5	156	4	42	60	151	152	154	156	158	194	211
5235935	159	11.5	156	4	43	64	149	152	154	156	158	195	213
5450827	159	12.3	156	4	42	69	150	152	154	156	158	194	219
5450827	159	11.9	156	4	45	60	150	152	154	156	158	197	210
5486221	158	9.8	156	3	34	65	151	152	155	156	158	186	216
5486221	159	13.0	156	4	46	68	150	152	154	156	158	198	218
5690571	159	11.7	157	4	42	69	150	153	155	157	159	195	219
5690571	159	11.4	156	4	45	64	150	152	154	156	158	197	214
5875079	159	12.3	156	4	11	78	150	153	155	156	159	164	228
5875079	159	10.8	157	3	31	71	150	153	155	157	158	184	221
5900945	158	9.3	156	3	8	64	151	153	155	156	158	161	215
5900945	159	11.1	157	3	10	79	149	153	155	157	158	163	228
5927368	159	12.1	156	4	45	76	151	152	154	156	158	197	227

5927368	159	10.9	156	3	34	64	151	153	155	156	158	187	215
6097084	159	11.2	156	3	9	80	151	153	155	156	158	162	231
6097084	159	11.2	157	3	12	73	150	153	155	157	158	165	223
6597011	158	7.2	157	4	9	75	152	153	155	157	159	162	227
6597011	158	9.2	157	3	9	72	151	153	156	157	159	162	223
6768571	159	10.7	157	3	9	76	151	154	156	157	159	163	227
6768571	158	8.1	157	4	9	67	151	154	155	157	159	163	218
6854107	159	9.1	158	4	9	77	152	154	156	158	160	163	229
6854107	159	11.1	157	4	10	84	152	153	155	157	159	163	236
6932837	160	11.5	158	3	11	90	151	153	156	158	159	164	241
6932837	159	10.4	157	3	10	84	151	153	156	157	159	163	235
7069638	158	6.7	157	3	8	82	151	154	156	157	159	162	233
7069638	160	12.0	158	3	12	82	152	154	156	158	159	166	234
7302069	159	11.1	158	4	9	90	152	154	156	158	160	163	242
7302069	159	7.7	158	4	9	85	152	154	156	158	160	163	237
7450186	159	8.4	158	4	9	87	152	154	156	158	160	163	239
7450186	159	10.9	158	4	9	90	153	154	156	158	160	163	243
8716744	160	3.1	159	5	10	18	151	155	157	159	162	165	169
8716744	160	6.1	159	3	9	94	152	155	158	159	161	164	246
9561827	160	3.1	160	4	11	18	152	155	158	160	162	166	170
9561827	160	3.1	160	4	11	17	153	156	158	160	162	167	170
9887400	161	7.7	160	4	11	120	153	155	158	160	162	166	273
9887400	161	9.1	160	5	11	112	154	156	158	160	163	167	266
9997495	160	3.1	160	4	10	17	152	155	158	160	162	165	169
9997495	161	3.1	161	4	10	16	153	156	159	161	163	166	169
12936965	163	3.7	163	5	12	19	155	158	161	163	166	170	174
12936965	164	3.8	163	5	13	19	155	157	161	163	166	170	174
12997436	163	3.6	163	5	11	20	155	158	161	163	166	169	175
12997436	163	3.7	163	6	12	21	154	158	160	163	166	170	175
13270592	163	3.6	163	5	12	19	154	158	161	163	166	170	173
13270592	164	3.4	164	5	12	16	157	158	161	164	166	170	173
13479768	164	3.4	164	4	11	20	153	159	162	164	166	170	173
13479768	164	3.4	163	5	11	18	156	159	161	163	166	170	174
13946417	164	3.4	164	5	11	19	158	159	162	164	167	170	177
13946417	164	3.5	164	5	11	19	155	159	162	164	167	170	174
14049511	164	3.8	164	5	12	23	154	159	162	164	167	171	177
14049511	164	3.7	164	5	13	20	155	158	162	164	167	171	175
15354841	165	3.6	165	4	12	20	158	160	163	165	167	172	178
15354841	166	3.7	166	5	12	20	156	160	163	166	168	172	176
16495367	166	3.6	166	5	13	18	157	160	164	166	169	173	175
16495367	166	4.2	167	5	13	25	153	160	164	167	169	173	178
16777216	167	3.9	167	5	13	23	157	160	164	167	169	173	180
16777216	167	4.0	166	5	13	22	158	161	164	166	169	174	180
16777216	167	4.2	167	6	15	25	157	159	164	167	170	174	182

A.2.2 Leaf Size Data

66	4	2.1	5	4	6	7	0	1	2	5	6	7	7
66	65	0.0	65	0	0	0	65	65	65	65	65	65	65

72	5	2.7	4	6	8	8	1	1	2	4	8	9	9
72	5	2.8	5	6	8	9	0	1	3	5	9	9	9
77	5	2.3	4	3	7	9	0	2	3	4	6	9	9
77	5	2.2	5	3	7	9	0	2	3	5	6	9	9
79	5	2.7	5	4	9	10	0	1	4	5	8	10	10
79	5	2.9	3	6	8	8	2	2	3	3	9	10	10
90	6	2.7	5	4	9	10	1	2	3	5	7	11	11
90	6	2.4	6	2	7	10	0	3	4	6	6	10	10
94	6	2.3	6	4	7	8	2	3	3	6	7	10	10
94	5	1.3	5	2	4	4	3	3	4	5	6	7	7
98	7	3.3	6	2	11	11	2	2	5	6	7	13	13
98	6	2.6	6	4	8	10	1	3	4	6	8	11	11
100	6	2.4	5	4	7	9	1	3	5	5	9	10	10
100	6	2.7	6	5	7	8	2	3	4	6	9	10	10
107	7	3.6	6	6	11	11	3	3	5	6	11	14	14
107	6	1.8	7	3	6	8	1	3	5	7	8	9	9
114	7	2.4	8	3	8	9	1	2	6	8	9	10	10
114	7	2.5	8	2	8	8	2	2	7	8	9	10	10
116	7	2.2	7	4	8	9	2	3	5	7	9	11	11
116	7	2.1	8	2	7	9	1	3	6	8	8	10	10
140	9	2.7	9	5	7	8	4	5	6	9	11	12	12
140	9	3.5	8	5	12	12	4	4	6	8	11	16	16
151	9	2.1	10	3	7	10	2	5	7	10	10	12	12
151	9	2.7	9	2	10	11	3	4	9	9	11	14	14
157	10	2.6	9	4	8	9	4	5	8	9	12	13	13
157	9	1.8	10	3	6	7	4	5	8	10	11	11	11
174	11	2.8	11	5	9	12	4	7	8	11	13	16	16
174	12	4.5	12	5	14	18	2	6	9	12	14	20	20
185	11	1.6	11	3	5	5	8	8	9	11	12	13	13
185	11	2.6	10	3	9	10	6	7	10	10	13	16	16
193	12	3.7	11	7	11	12	6	7	10	11	17	18	18
193	12	3.0	11	5	10	10	6	6	10	11	15	16	16
216	13	3.5	14	6	11	11	8	8	11	14	17	19	19
216	14	3.9	12	6	14	15	6	7	11	12	17	21	21
239	15	4.1	15	6	16	16	7	7	12	15	18	23	23
239	15	3.4	14	6	10	11	8	9	13	14	19	19	19
239	15	4.0	15	6	13	14	9	10	12	15	18	23	23
239	15	3.7	15	6	13	13	8	8	12	15	18	21	21
256	16	3.3	17	5	9	11	10	12	12	17	17	21	21
256	16	4.0	15	8	12	13	10	11	13	15	21	23	23
256	16	3.2	16	3	12	13	8	9	14	16	17	21	21
256	16	3.5	17	5	12	13	8	9	13	17	18	21	21
256	16	4.5	16	7	17	17	7	7	13	16	20	24	24
270	16	2.8	17	5	9	10	10	11	14	17	19	20	20
270	17	3.9	17	7	12	15	8	11	14	17	21	23	23
283	18	5.3	17	9	18	18	8	8	14	17	23	26	26
283	17	3.3	18	4	10	13	9	12	16	18	20	22	22
289	18	3.8	19	6	12	13	9	10	15	19	21	22	22
289	18	3.9	18	5	13	15	11	13	15	18	20	26	26
306	19	4.3	18	5	15	16	12	13	17	18	22	28	28
306	19	3.3	18	4	12	14	11	13	17	18	21	25	25
325	20	3.4	21	5	10	11	13	14	18	21	23	24	24
325	20	4.1	19	5	14	16	12	14	18	19	23	28	28
327	18	5.7	20	5	23	24	0	1	16	20	21	24	24

327	20	4.0	21	6	14	14	13	13	17	21	23	27	27
346	22	4.4	22	5	15	16	14	15	19	22	24	30	30
346	21	4.2	22	6	14	15	13	14	19	22	25	28	28
348	20	5.9	21	4	24	27	0	3	18	21	22	27	27
348	22	4.1	22	5	14	16	12	14	19	22	24	28	28
356	19	7.7	20	9	29	30	0	1	17	20	26	30	30
356	22	4.8	21	6	17	18	15	16	19	21	25	33	33
405	22	6.8	23	5	26	28	0	2	21	23	26	28	28
405	23	9.4	24	7	38	39	0	1	21	24	28	39	39
413	25	3.8	24	5	12	12	21	21	22	24	27	33	33
413	24	8.2	26	5	35	36	0	1	24	26	29	36	36
421	20	10.7	22	9	35	35	0	0	17	22	26	35	35
421	27	5.9	26	9	19	27	11	19	23	26	32	38	38
465	22	10.7	24	6	36	37	0	1	21	24	27	37	37
465	27	8.1	28	7	31	33	0	2	25	28	32	33	33
492	28	8.2	29	6	35	37	0	2	26	29	32	37	37
492	28	8.8	28	7	36	38	0	2	25	28	32	38	38
512	30	8.8	34	12	35	38	0	3	25	34	37	38	38
512	29	9.5	31	7	42	45	0	3	27	31	34	45	45
532	34	6.6	33	9	23	23	22	22	30	33	39	45	45
532	28	11.5	31	5	41	42	0	1	28	31	33	42	42
540	29	10.2	32	7	37	39	0	2	29	32	36	39	39
540	28	10.7	31	9	38	40	0	2	27	31	36	40	40
542	25	14.3	33	31	40	41	0	1	3	33	34	41	41
542	31	8.6	32	5	39	44	0	5	29	32	34	44	44
550	32	9.7	32	9	42	44	0	2	29	32	38	44	44
550	28	12.0	34	7	35	36	0	1	29	34	36	36	36
571	26	15.1	32	15	43	44	0	1	23	32	38	44	44
571	30	13.0	35	11	45	47	0	2	26	35	37	47	47
589	28	14.7	34	10	42	43	0	1	27	34	37	43	43
589	33	12.1	33	9	48	49	1	2	30	33	39	50	50
611	31	14.2	35	13	47	49	0	2	28	35	41	49	49
611	26	16.6	34	35	42	43	0	1	3	34	38	43	43
612	25	17.1	34	37	45	46	0	1	3	34	40	46	46
612	25	17.3	33	36	43	44	0	1	3	33	39	44	44
613	29	16.2	36	15	52	53	0	1	23	36	38	53	53
613	33	13.4	37	10	45	46	0	1	31	37	41	46	46
645	30	17.4	36	37	52	53	0	1	5	36	42	53	53
645	24	17.4	28	36	45	46	0	1	3	28	39	46	46
657	30	17.7	37	39	48	49	0	1	5	37	44	49	49
657	26	17.1	37	35	44	45	0	1	4	37	39	45	45
709	26	18.9	36	41	47	48	0	1	3	36	44	48	48
709	26	20.3	41	41	50	51	0	1	3	41	44	51	51
897	26	24.6	6	50	60	61	0	1	3	6	53	61	61
897	17	21.1	5	40	60	61	0	1	3	5	43	61	61
922	27	26.8	6	53	63	64	0	1	3	6	56	64	64
922	17	23.4	5	7	63	64	0	1	3	5	10	64	64
990	17	23.9	5	5	70	71	0	1	3	5	8	71	71
990	14	21.9	4	4	64	66	0	1	3	4	7	65	66
1024	26	26.3	7	51	67	69	0	2	4	7	55	69	69
1024	9	15.7	4	3	57	58	0	1	3	4	6	58	58
1024	20	25.4	5	47	65	66	0	1	3	5	50	66	66
1039	14	20.8	4	4	60	61	0	1	3	4	7	61	61
1039	25	27.2	5	53	60	70	1	2	3	5	56	62	71

1062	24	28.6	5	61	67	69	0	1	4	5	65	68	69
1062	15	21.4	5	5	59	60	0	1	3	5	8	60	60
1269	9	14.7	5	4	63	65	0	2	3	5	7	65	65
1269	14	25.4	5	5	89	90	0	1	3	5	8	90	90
1400	8	12.8	5	3	10	78	0	2	4	5	7	12	78
1400	11	19.5	5	3	81	83	0	2	4	5	7	83	83
1409	10	19.3	5	3	85	87	0	2	4	5	7	87	87
1409	9	17.1	5	3	77	80	0	3	4	5	7	80	80
1544	15	26.3	6	4	94	95	1	2	4	6	8	96	96
1544	6	2.7	6	3	9	14	0	2	4	6	7	11	14
1597	11	20.5	6	3	90	92	1	3	5	6	8	93	93
1597	6	2.3	6	3	6	11	1	3	5	6	8	9	12
1638	12	23.8	6	5	98	101	0	3	4	6	9	101	101
1638	7	2.4	6	3	9	15	1	3	5	6	8	12	16
1695	7	2.4	6	3	8	12	1	3	5	6	8	11	13
1695	7	2.9	6	3	10	14	1	2	5	6	8	12	15
1787	7	2.9	7	4	9	13	1	3	5	7	9	12	14
1787	7	2.4	7	4	8	10	2	3	5	7	9	11	12
1795	7	2.8	7	4	9	15	0	3	5	7	9	12	15
1795	7	3.0	7	4	10	14	1	2	5	7	9	12	15
1871	7	2.6	7	4	8	15	1	3	5	7	9	11	16
1871	7	2.5	7	3	8	14	1	3	6	7	9	11	15
1947	8	2.7	7	3	9	13	3	4	6	7	9	13	16
1947	15	27.9	7	4	116	118	1	3	6	7	10	119	119
2016	7	2.5	7	3	9	13	2	4	6	7	9	13	15
2016	8	2.7	8	4	9	14	2	4	6	8	10	13	16
2061	8	3.1	8	4	11	16	1	3	6	8	10	14	17
2061	8	2.5	8	4	8	16	2	4	6	8	10	12	18
2397	10	3.2	10	5	11	17	2	5	7	10	12	16	19
2397	9	3.3	9	5	11	18	1	4	7	9	12	15	19
2403	9	3.1	9	4	10	19	1	5	7	9	11	15	20
2403	9	3.1	9	5	10	16	1	5	7	9	12	15	17
2555	10	2.7	10	4	9	12	4	5	8	10	12	14	16
2555	10	2.9	10	4	10	16	1	6	8	10	12	16	17
2760	11	3.1	11	5	10	14	4	6	8	11	13	16	18
2760	11	3.7	10	6	12	19	3	5	8	10	14	17	22
2790	11	3.4	11	4	11	18	2	5	9	11	13	16	20
2790	11	3.3	11	4	10	19	2	6	9	11	13	16	21
2825	11	3.1	11	4	10	15	4	6	9	11	13	16	19
2825	11	2.8	11	4	10	15	3	6	9	11	13	16	18
2971	12	3.4	12	5	12	17	3	6	9	12	14	18	20
2971	12	3.3	11	5	12	14	5	6	9	11	14	18	19
2996	12	3.8	12	5	12	20	2	6	9	12	14	18	22
2996	12	3.5	12	4	12	19	4	6	10	12	14	18	23
3096	12	3.6	11	6	12	15	4	6	9	11	15	18	19
3096	12	3.6	12	4	11	19	4	7	10	12	14	18	23
3224	12	3.4	12	5	11	17	5	7	10	12	15	18	22
3224	12	3.4	12	5	12	19	3	7	10	12	15	19	22
3878	15	4.1	15	6	13	20	5	9	12	15	18	22	25
3878	15	3.7	15	4	14	17	7	9	13	15	17	23	24
4096	16	3.9	16	6	12	21	4	10	13	16	19	22	25
4096	16	3.9	16	5	13	21	5	9	13	16	18	22	26
4096	16	3.8	16	6	11	21	6	11	13	16	19	22	27
4602	18	4.1	18	5	14	23	8	11	16	18	21	25	31

4602	18	4.4	18	6	14	26	6	11	15	18	21	25	32
4967	20	4.2	20	5	14	21	9	13	17	20	22	27	30
4967	19	4.9	18	6	16	34	0	11	16	18	22	27	34
5000	20	4.5	19	6	15	21	9	12	17	19	23	27	30
5000	19	3.9	19	6	12	20	9	13	16	19	22	25	29
5130	21	4.7	21	7	16	22	9	12	18	21	25	28	31
5130	20	5.0	20	6	19	24	9	12	17	20	23	31	33
5156	20	4.7	20	8	14	31	1	13	16	20	24	27	32
5156	20	5.2	20	6	16	33	0	12	17	20	23	28	33
5161	20	5.2	20	6	17	31	1	13	17	20	23	30	32
5161	20	4.8	19	6	16	34	1	13	17	19	23	29	35
5375	21	5.0	21	7	16	23	10	13	18	21	25	29	33
5375	21	4.7	21	6	16	26	8	13	18	21	24	29	34
5402	21	5.1	21	7	16	33	0	13	17	21	24	29	33
5402	21	4.8	20	5	14	34	0	13	18	20	23	27	34
5454	21	4.8	21	7	16	38	1	14	18	21	25	30	39
5454	20	5.9	21	5	19	39	0	10	18	21	23	29	39
6545	25	7.1	26	7	23	38	0	12	21	26	28	35	38
6545	24	6.7	25	7	20	37	0	14	21	25	28	34	37
6752	25	6.8	26	8	18	39	1	17	22	26	30	35	40
6752	26	6.8	27	8	20	40	1	16	23	27	31	36	41
6800	25	7.9	26	7	34	44	0	3	23	26	30	37	44
6800	26	6.5	26	6	18	37	1	16	23	26	29	34	38
7144	26	8.1	27	8	34	42	0	2	23	27	31	36	42
7144	26	6.8	27	7	19	43	0	17	23	27	30	36	43
7173	26	7.9	27	7	34	46	0	2	23	27	30	36	46
7173	27	7.0	28	8	21	41	1	17	24	28	32	38	42
7290	27	8.2	28	9	33	41	1	4	23	28	32	37	42
7290	26	7.9	28	9	33	39	0	3	23	28	32	36	39
7866	29	9.2	29	9	39	46	0	2	25	29	34	41	46
7866	29	9.8	30	8	38	47	0	2	26	30	34	40	47
8175	28	10.6	29	8	40	43	0	2	26	29	34	42	43
8175	28	10.6	30	9	39	44	0	2	26	30	35	41	44
8668	28	12.6	31	9	41	49	0	1	27	31	36	42	49
8668	29	12.0	33	9	39	46	0	2	28	33	37	41	46
9238	28	13.9	33	10	44	48	0	1	27	33	37	45	48
9238	27	14.3	32	13	41	45	0	1	23	32	36	42	45
10525	31	17.0	39	22	46	54	0	1	21	39	43	47	54
10525	27	18.3	35	37	49	53	0	1	4	35	41	50	53
10540	28	17.4	36	36	48	53	0	1	5	36	41	49	53
10540	29	18.3	37	39	50	57	0	1	4	37	43	51	57
12535	25	22.4	36	43	52	61	0	1	3	36	46	53	61
12535	20	21.3	5	41	52	59	0	1	2	5	43	53	59
12969	22	22.5	5	43	55	64	0	1	3	5	46	56	64
12969	23	22.8	6	43	56	65	0	1	3	6	46	57	65
14133	23	24.4	5	46	61	83	0	1	3	5	49	62	83
14133	25	24.7	6	48	59	67	0	1	3	6	51	60	67
15298	22	26.4	5	51	65	76	0	1	3	5	54	66	76
15298	17	23.5	4	6	62	68	0	1	3	4	9	63	68
15464	20	24.8	5	49	62	71	0	1	3	5	52	63	71
15464	18	23.7	5	42	61	78	0	1	3	5	45	62	78
15768	21	25.7	5	49	64	74	0	1	3	5	52	65	74
15768	17	23.4	4	40	61	69	0	1	3	4	43	62	69
16384	16	23.7	5	5	64	79	0	1	3	5	8	65	79

16384	19	25.0	5	47	64	77	0	1	3	5	50	65	77
16384	19	25.5	5	45	66	80	0	1	3	5	48	67	80
17684	18	25.7	5	4	71	86	0	1	4	5	8	72	86
17684	14	22.8	5	4	70	83	0	1	3	5	7	71	83
19226	9	16.7	5	3	60	78	0	2	3	5	6	62	78
19226	14	23.2	5	4	70	83	1	2	3	5	7	72	84
20849	10	17.7	5	3	66	85	0	2	4	5	7	68	85
20849	11	19.9	5	3	72	98	0	2	4	5	7	74	98
21442	10	18.5	5	3	74	97	0	2	4	5	7	76	97
21442	10	18.6	5	4	71	87	0	2	3	5	7	73	87
22342	9	16.1	6	3	15	86	0	2	4	6	7	17	86
22342	9	17.1	5	3	65	95	1	2	4	5	7	67	96
23202	8	15.1	6	3	9	100	0	2	4	6	7	11	100
23202	9	15.5	6	3	11	91	1	2	4	6	7	13	92
24436	9	15.9	6	4	9	92	1	2	4	6	8	11	93
24436	8	12.4	6	4	9	88	1	2	4	6	8	11	89
25278	8	11.9	6	4	9	97	0	2	4	6	8	11	97
25278	11	19.7	6	3	76	108	1	2	5	6	8	78	109
26466	8	10.7	7	3	8	105	0	3	5	7	8	11	105
26466	10	16.6	6	3	10	103	2	3	5	6	8	13	105
26496	7	10.1	6	3	8	111	0	3	5	6	8	11	111
26496	8	11.7	6	3	8	104	1	3	5	6	8	11	105
26894	10	17.3	7	4	10	114	1	2	5	7	9	12	115
26894	8	12.2	6	3	9	107	1	3	5	6	8	12	108
27865	8	10.1	7	3	9	116	1	3	5	7	8	12	117
27865	8	11.1	6	4	8	118	1	3	5	6	9	11	119
28676	8	10.4	7	4	11	103	1	2	5	7	9	13	104
28676	8	8.5	7	4	9	108	1	3	5	7	9	12	109
28823	8	10.4	7	4	8	107	1	3	5	7	9	11	108
28823	8	11.5	7	4	9	117	1	4	5	7	9	13	118
33146	11	18.8	8	4	9	150	1	4	6	8	10	13	151
33146	9	6.5	8	4	9	99	2	4	6	8	10	13	101
33494	9	8.2	8	4	9	129	2	4	6	8	10	13	131
33494	8	2.8	8	4	8	17	1	4	6	8	10	12	18
33855	8	3.0	8	4	10	18	0	4	6	8	10	14	18
33855	10	13.1	8	3	10	146	1	4	7	8	10	14	147
37192	9	3.1	9	4	10	17	2	4	7	9	11	14	19
37192	10	3.1	9	5	10	18	1	5	7	9	12	15	19
38763	9	2.7	9	4	9	15	3	5	7	9	11	14	18
38763	10	8.6	9	4	11	136	2	4	7	9	11	15	138
38908	10	3.1	10	5	10	15	3	5	7	10	12	15	18
38908	10	3.1	9	5	10	16	1	5	7	9	12	15	17
40188	11	13.1	10	4	12	152	2	5	8	10	12	17	154
40188	10	3.0	9	5	10	15	3	5	7	9	12	15	18
41640	11	3.1	10	3	11	18	2	5	9	10	12	16	20
41640	10	3.2	10	4	10	19	3	6	8	10	12	16	22
43155	11	3.2	11	4	11	19	1	6	9	11	13	17	20
43155	11	3.2	11	4	10	17	3	6	9	11	13	16	20
44845	11	3.4	11	5	11	17	3	6	9	11	14	17	20
44845	11	3.3	11	4	11	19	3	6	9	11	13	17	22
46825	12	3.4	11	5	11	18	4	6	9	11	14	17	22
46825	11	3.4	11	5	11	18	4	6	9	11	14	17	22
50226	12	3.4	12	4	11	20	3	7	10	12	14	18	23
50226	12	3.9	13	5	12	23	2	6	10	13	15	18	25

62700	15	4.0	15	5	13	22	2	9	13	15	18	22	24
62700	15	3.9	15	6	13	20	6	8	12	15	18	21	26
64497	16	4.4	16	5	14	27	4	10	13	16	18	24	31
64497	16	3.8	16	5	13	19	7	9	13	16	18	22	26
65536	16	3.8	16	6	13	19	7	9	13	16	19	22	26
65536	16	4.0	16	6	14	24	3	9	13	16	19	23	27
65536	16	4.0	16	6	13	23	7	10	13	16	19	23	30
67433	17	4.0	17	5	14	24	7	10	14	17	19	24	31
67433	16	4.1	16	6	13	24	6	10	13	16	19	23	30
78442	19	4.4	19	5	14	33	0	13	17	19	22	27	33
78442	18	4.6	18	6	14	30	1	11	15	18	21	25	31
81254	19	5.0	19	5	15	42	0	12	17	19	22	27	42
81254	20	4.6	20	6	15	26	10	12	17	20	23	27	36
88955	21	4.8	21	6	16	32	1	13	18	21	24	29	33
88955	22	5.0	22	6	16	35	1	14	19	22	25	30	36
92905	22	5.3	23	7	17	35	0	14	19	23	26	31	35
92905	22	5.4	23	7	17	33	0	14	19	23	26	31	33
93623	23	4.4	23	6	14	34	0	16	20	23	26	30	34
93623	23	5.7	22	7	16	38	0	15	20	22	27	31	38
94561	23	6.2	23	6	19	41	0	13	20	23	26	32	41
94561	22	5.3	23	7	15	35	0	15	19	23	26	30	35
95372	23	5.4	24	7	16	36	1	15	20	24	27	31	37
95372	22	5.9	23	6	17	33	0	14	20	23	26	31	33
99160	24	6.1	24	6	18	44	0	15	21	24	27	33	44
99160	23	6.5	23	8	18	40	0	14	20	23	28	32	40
101225	24	5.9	24	6	16	43	0	16	21	24	27	32	43
101225	24	6.1	25	6	18	40	0	16	22	25	28	34	40
101559	24	6.8	25	6	22	40	0	11	22	25	28	33	40
101559	24	6.5	25	7	18	40	0	16	21	25	28	34	40
105691	25	6.1	25	8	17	36	1	17	21	25	29	34	37
105691	25	6.0	26	6	18	42	0	17	22	26	28	35	42
110967	25	7.8	26	7	33	40	0	2	22	26	29	35	40
110967	26	6.5	27	6	17	44	1	18	24	27	30	35	45
111404	26	7.5	27	7	31	40	1	4	23	27	30	35	41
111404	25	6.9	26	6	23	38	0	11	23	26	29	34	38
114879	26	8.0	28	7	34	44	1	3	24	28	31	37	45
114879	27	7.1	28	7	21	45	0	15	24	28	31	36	45
129414	29	9.8	30	8	38	52	0	2	26	30	34	40	52
129414	28	10.1	30	8	38	48	0	2	26	30	34	40	48
132549	29	10.8	31	9	39	50	0	2	26	31	35	41	50
132549	29	9.9	31	7	38	48	0	2	27	31	34	40	48
135777	28	11.5	31	10	41	47	0	2	26	31	36	43	47
135777	28	11.9	31	9	40	49	0	1	26	31	35	41	49
137216	29	12.0	32	9	42	47	0	1	27	32	36	43	47
137216	28	12.6	32	9	41	51	0	1	27	32	36	42	51
139718	28	12.4	32	9	41	49	0	1	27	32	36	42	49
139718	28	12.8	32	9	40	50	0	1	27	32	36	41	50
149844	29	14.8	34	12	45	52	0	1	26	34	38	46	52
149844	29	14.7	35	12	44	49	0	1	27	35	39	45	49
155579	30	14.2	36	12	43	56	0	2	28	36	40	45	56
155579	30	15.2	35	11	46	59	0	1	28	35	39	47	59
168788	29	17.5	37	37	48	57	0	1	5	37	42	49	57
168788	31	17.5	37	37	49	62	0	1	6	37	43	50	62
180324	28	19.8	37	40	51	57	0	1	4	37	44	52	57

180324	27	19.6	36	40	50	59	0	1	4	36	44	51	59
182842	25	20.4	37	41	49	59	0	1	3	37	44	50	59
182842	24	20.0	33	41	48	56	0	1	2	33	43	49	56
186034	25	20.7	34	40	52	59	0	1	3	34	43	53	59
186034	28	20.3	38	42	50	62	0	1	3	38	45	51	62
187594	28	20.3	37	42	51	59	0	1	3	37	45	52	59
187594	25	21.3	34	41	53	65	0	1	3	34	44	54	65
193543	26	21.8	37	44	53	63	0	1	3	37	47	54	63
193543	24	22.1	7	43	54	63	0	1	3	7	46	55	63
211998	25	23.6	6	45	58	76	0	1	3	6	48	59	76
211998	24	23.3	6	45	56	67	0	1	3	6	48	57	67
212668	22	23.7	5	45	57	64	0	1	3	5	48	58	64
212668	25	23.5	8	46	57	69	0	1	3	8	49	58	69
220475	19	22.8	5	43	57	66	0	1	3	5	46	58	66
220475	21	23.6	5	45	58	67	0	1	3	5	48	59	67
223090	21	23.9	5	47	58	69	0	1	3	5	50	59	69
223090	23	24.3	5	47	60	65	0	1	3	5	50	61	65
227163	22	24.0	5	46	60	71	0	1	3	5	49	61	71
227163	19	23.4	4	47	58	72	0	1	2	4	49	59	72
234239	21	25.2	5	48	61	85	0	1	3	5	51	62	85
234239	24	25.3	6	49	63	74	0	1	3	6	52	64	74
248714	21	25.6	5	48	65	74	0	1	3	5	51	66	74
248714	22	25.9	5	51	63	75	0	1	3	5	54	64	75
252247	18	24.7	5	44	66	79	0	1	3	5	47	67	79
252247	21	26.1	5	50	68	79	0	1	3	5	53	69	79
256849	20	25.7	5	48	66	79	0	1	3	5	51	67	79
256849	16	23.3	5	4	64	75	0	1	3	5	7	65	75
257794	18	23.9	5	45	64	70	0	1	3	5	48	65	70
257794	17	24.0	4	5	65	75	0	1	3	4	8	66	75
262144	16	23.1	5	5	62	75	0	2	3	5	8	64	75
262144	17	24.6	5	6	67	81	0	1	3	5	9	68	81
262144	15	22.7	4	4	65	79	1	1	3	4	7	66	80
282691	17	24.6	5	5	68	83	0	1	3	5	8	69	83
282691	14	23.1	5	4	72	80	0	1	3	5	7	73	80
292185	15	24.0	5	4	71	93	0	1	3	5	7	72	93
292185	17	25.5	5	5	71	83	0	2	3	5	8	73	83
327699	14	23.2	6	4	75	100	0	1	4	6	8	76	100
327699	10	18.7	5	4	70	84	0	2	3	5	7	72	84
333128	9	17.6	5	4	67	103	0	2	3	5	7	69	103
333128	10	18.3	5	3	72	95	0	2	4	5	7	74	95
370321	10	18.1	6	4	56	106	1	2	4	6	8	58	107
370321	12	22.5	6	4	84	99	0	2	4	6	8	86	99
376186	10	17.7	6	4	11	93	1	2	4	6	8	13	94
376186	10	17.4	6	3	71	93	1	2	4	6	7	73	94
376561	8	15.9	5	3	9	105	0	2	4	5	7	11	105
376561	9	16.8	6	3	10	95	0	3	4	6	7	13	95
396823	9	14.6	6	3	8	99	1	3	5	6	8	11	100
396823	9	15.3	6	3	9	105	1	3	5	6	8	12	106
414618	8	11.3	6	3	7	105	1	3	5	6	8	10	106
414618	8	11.3	6	3	9	106	1	2	5	6	8	11	107
415024	10	18.7	6	3	10	107	1	3	5	6	8	13	108
415024	9	15.5	6	4	10	116	0	2	5	6	9	12	116
456137	8	12.4	7	4	9	114	1	3	5	7	9	12	115
456137	9	14.6	7	4	9	117	1	3	5	7	9	12	118

464041	7	5.9	7	4	9	89	1	3	5	7	9	12	90
464041	9	10.8	7	3	10	106	2	3	6	7	9	13	108
489220	8	6.4	7	3	9	101	0	3	6	7	9	12	101
489220	8	6.5	7	3	9	100	2	3	6	7	9	12	102
522924	8	7.4	8	4	9	116	0	4	6	8	10	13	116
522924	8	7.6	8	4	10	119	2	3	6	8	10	13	121
526827	8	2.9	8	4	9	17	1	4	6	8	10	13	18
526827	10	12.1	8	4	10	132	2	4	6	8	10	14	134
556524	9	2.9	8	5	9	18	0	4	6	8	11	13	18
556524	9	8.3	9	4	10	131	2	4	7	9	11	14	133
594179	9	3.1	9	4	10	15	3	4	7	9	11	14	18
594179	9	3.1	9	4	10	17	2	4	7	9	11	14	19
601537	9	3.0	9	4	10	17	2	5	7	9	11	15	19
601537	9	3.1	9	4	10	16	2	4	7	9	11	14	18
623975	10	3.0	10	4	11	16	3	5	8	10	12	16	19
623975	10	8.4	9	4	9	134	1	5	7	9	11	14	135
633731	10	3.0	10	3	10	19	2	5	8	10	11	15	21
633731	10	3.0	9	4	10	16	2	5	7	9	11	15	18
650258	10	3.1	10	4	10	16	3	5	8	10	12	15	19
650258	10	3.2	10	4	10	21	2	5	8	10	12	15	23
675966	10	3.1	10	4	11	16	3	5	8	10	12	16	19
675966	10	3.2	10	5	11	18	2	5	8	10	13	16	20
711698	11	3.3	11	4	11	18	3	6	9	11	13	17	21
711698	11	3.2	11	5	10	15	3	6	8	11	13	16	18
815935	13	3.8	12	5	13	23	2	6	10	12	15	19	25
815935	12	3.3	12	5	11	17	4	7	10	12	15	18	21
878307	13	3.7	13	5	13	20	4	7	11	13	16	20	24
878307	13	3.5	13	5	12	18	4	8	11	13	16	20	22
986780	15	4.1	15	5	14	24	3	8	12	15	17	22	27
986780	15	3.7	15	5	13	25	0	9	13	15	18	22	25
1008012	15	4.0	15	6	12	22	5	9	12	15	18	21	27
1008012	15	3.9	15	5	13	22	6	9	13	15	18	22	28
1048576	16	3.9	16	6	13	19	8	10	13	16	19	23	27
1048576	16	3.8	16	5	13	22	7	10	13	16	18	23	29
1048576	16	4.1	15	5	14	22	6	9	13	15	18	23	28
1069151	16	3.9	16	5	12	33	0	10	14	16	19	22	33
1069151	16	4.2	16	6	13	25	1	9	13	16	19	22	26
1113665	17	3.9	17	5	13	21	8	11	14	17	19	24	29
1113665	17	4.1	17	6	14	28	1	10	14	17	20	24	29
1128563	17	3.9	17	5	12	25	3	11	14	17	19	23	28
1128563	17	4.3	17	6	15	21	7	10	14	17	20	25	28
1192316	18	4.2	18	6	13	25	7	11	15	18	21	24	32
1192316	18	4.3	18	6	14	29	1	11	15	18	21	25	30
1368818	21	4.8	21	5	15	31	0	13	18	21	23	28	31
1368818	21	4.6	21	7	15	22	10	14	17	21	24	29	32
1449558	22	4.7	22	6	15	33	2	15	19	22	25	30	35
1449558	22	5.1	22	7	17	35	2	14	19	22	26	31	37
1463861	22	5.2	22	6	16	34	0	14	19	22	25	30	34
1463861	22	4.7	22	7	15	35	0	15	19	22	26	30	35
1781937	25	8.2	26	8	33	41	0	3	22	26	30	36	41
1781937	25	7.6	26	7	31	41	0	4	23	26	30	35	41
1785742	26	7.4	27	7	21	52	0	14	23	27	30	35	52
1785742	25	7.2	26	8	24	43	0	11	22	26	30	35	43
1884161	26	9.5	28	8	35	42	0	2	24	28	32	37	42

1884161	27	8.0	28	7	34	44	0	3	24	28	31	37	44
2061614	29	9.3	31	8	38	43	0	2	26	31	34	40	43
2061614	29	9.0	31	8	36	48	1	3	26	31	34	39	49
2366039	31	12.2	34	8	42	51	0	1	30	34	38	43	51
2366039	28	14.8	33	13	44	58	0	1	25	33	38	45	58
2392176	29	14.1	34	11	43	50	0	1	27	34	38	44	50
2392176	27	15.6	33	33	44	51	0	1	5	33	38	45	51
2417864	28	15.4	34	14	45	52	0	1	24	34	38	46	52
2417864	30	13.6	34	10	45	53	0	1	29	34	39	46	53
2525166	29	15.5	35	15	45	51	0	1	24	35	39	46	51
2525166	29	16.0	35	20	45	56	0	1	19	35	39	46	56
2672788	27	18.0	35	37	47	61	0	1	4	35	41	48	61
2672788	28	18.0	37	37	47	57	0	1	4	37	41	48	57
2861570	29	19.2	38	39	50	60	0	1	4	38	43	51	60
2861570	29	18.8	39	40	48	56	0	1	4	39	44	49	56
2884139	25	20.0	35	39	50	64	0	1	3	35	42	51	64
2884139	28	19.8	38	41	51	56	0	1	4	38	45	52	56
3184126	25	22.1	34	43	54	61	0	1	3	34	46	55	61
3184126	26	22.5	37	45	54	65	0	1	3	37	48	55	65
3306465	25	22.9	7	45	55	65	0	1	3	7	48	56	65
3306465	27	23.0	41	45	56	67	0	1	3	41	48	57	67
3508605	21	24.2	5	46	58	73	0	1	3	5	49	59	73
3508605	24	23.8	6	46	59	65	0	1	3	6	49	60	65
3663174	23	24.6	5	47	61	69	0	1	3	5	50	62	69
3663174	22	24.5	5	47	61	71	0	1	3	5	50	62	71
3703674	23	24.9	5	48	61	69	0	1	3	5	51	62	69
3703674	21	24.2	4	46	60	71	0	1	3	4	49	61	71
3758810	25	25.6	6	48	62	76	0	1	3	6	51	63	76
3758810	21	24.4	5	48	61	72	0	1	3	5	51	62	72
3878401	19	24.8	5	48	64	76	0	1	3	5	51	65	76
3878401	19	24.4	5	48	61	67	0	1	3	5	51	62	67
3916543	20	25.2	5	49	63	73	0	1	3	5	52	64	73
3916543	21	25.8	5	48	66	77	0	1	3	5	51	67	77
4176245	16	23.6	5	5	65	77	0	1	3	5	8	66	77
4176245	15	22.4	4	4	63	73	0	1	3	4	7	64	73
4194304	15	23.1	4	4	66	75	0	1	3	4	7	67	75
4194304	16	23.1	4	5	63	86	0	1	3	4	8	64	86
4194304	17	23.8	5	6	64	74	0	1	3	5	9	65	74
4263816	14	21.8	5	4	63	81	0	1	3	5	7	64	81
4263816	14	21.9	5	4	64	77	0	1	3	5	7	65	77
4467515	16	24.1	5	5	69	78	0	1	3	5	8	70	78
4467515	16	24.5	5	4	69	86	0	1	3	5	7	70	86
4777900	12	20.6	5	3	68	90	0	1	3	5	6	69	90
4777900	15	24.1	5	3	71	87	0	2	4	5	7	73	87
4874035	11	19.6	5	3	70	82	0	2	4	5	7	72	82
4874035	15	23.7	5	3	71	88	0	2	4	5	7	73	88
5226162	12	20.3	5	3	69	89	0	2	4	5	7	71	89
5226162	13	21.5	6	3	74	88	1	2	4	6	7	76	89
5235935	11	19.4	5	3	71	90	0	2	4	5	7	73	90
5235935	10	18.9	5	3	72	92	0	2	4	5	7	74	92
5450827	11	20.4	5	3	71	98	0	2	4	5	7	73	98
5450827	11	19.9	5	3	74	89	0	2	4	5	7	76	89
5486221	9	16.1	5	3	63	94	1	2	4	5	7	65	95
5486221	11	20.8	5	3	75	97	0	2	4	5	7	77	97

5690571	11	19.0	6	4	72	98	0	2	4	6	8	74	98
5690571	10	18.6	5	3	74	93	0	2	4	5	7	76	93
5875079	10	18.4	6	4	11	107	0	2	4	6	8	13	107
5875079	10	17.1	6	4	61	100	0	2	4	6	8	63	100
5900945	8	14.8	5	3	9	93	1	2	4	5	7	11	94
5900945	9	16.9	6	3	10	106	1	2	4	6	7	12	107
5927368	10	18.7	5	4	74	106	0	2	4	5	8	76	106
5927368	10	17.3	5	4	64	94	0	2	4	5	8	66	94
6097084	9	16.8	6	3	9	110	0	2	4	6	7	11	110
6097084	10	17.3	6	4	12	102	0	2	4	6	8	14	102
6597011	7	10.2	6	4	9	105	1	2	4	6	8	11	106
6597011	8	13.6	6	3	8	101	1	3	5	6	8	11	102
6768571	9	15.7	7	3	9	105	1	3	5	7	8	12	106
6768571	8	12.1	6	4	9	96	1	3	4	6	8	12	97
6854107	9	13.1	7	4	10	107	1	3	5	7	9	13	108
6854107	9	16.2	7	3	10	114	1	2	5	7	8	12	115
6932837	10	16.6	7	4	10	119	1	3	5	7	9	13	120
6932837	9	15.2	6	3	10	113	1	2	5	6	8	12	114
7069638	7	9.2	6	4	8	112	0	3	5	6	9	11	112
7069638	10	17.6	7	3	12	112	1	3	5	7	8	15	113
7302069	9	15.1	7	4	9	120	1	3	5	7	9	12	121
7302069	8	10.7	7	3	9	114	2	3	6	7	9	12	116
7450186	8	11.5	7	4	9	117	1	3	5	7	9	12	118
7450186	9	15.0	7	4	9	120	2	3	5	7	9	12	122
8716744	9	3.0	8	5	10	18	0	4	6	8	11	14	18
8716744	9	7.8	8	3	9	124	1	4	7	8	10	13	125
9561827	9	3.1	9	4	11	18	1	4	7	9	11	15	19
9561827	9	3.1	9	4	11	17	2	5	7	9	11	16	19
9887400	10	9.5	9	4	11	150	2	4	7	9	11	15	152
9887400	10	11.6	9	5	11	142	3	5	7	9	12	16	145
9997495	9	3.1	9	4	10	17	1	4	7	9	11	14	18
9997495	10	3.1	10	4	10	16	2	5	8	10	12	15	18
12936965	12	3.7	12	5	12	19	4	7	10	12	15	19	23
12936965	13	3.8	12	5	13	19	4	6	10	12	15	19	23
12997436	12	3.6	12	5	11	20	4	7	10	12	15	18	24
12997436	12	3.7	12	6	12	21	3	7	9	12	15	19	24
13270592	12	3.6	12	5	12	19	3	7	10	12	15	19	22
13270592	13	3.4	13	5	12	16	6	7	10	13	15	19	22
13479768	13	3.4	13	4	11	20	2	8	11	13	15	19	22
13479768	13	3.4	12	5	11	18	5	8	10	12	15	19	23
13946417	13	3.4	13	5	11	19	7	8	11	13	16	19	26
13946417	13	3.5	13	5	11	19	4	8	11	13	16	19	23
14049511	13	3.8	13	5	12	23	3	8	11	13	16	20	26
14049511	13	3.8	13	5	13	24	0	7	11	13	16	20	24
15354841	14	3.6	14	4	12	20	7	9	12	14	16	21	27
15354841	14	3.8	15	5	12	23	2	9	12	15	17	21	25
16495367	15	3.6	15	5	13	18	6	9	13	15	18	22	24
16495367	15	4.2	16	5	14	26	1	8	13	16	18	22	27
16777216	16	3.9	16	5	13	23	6	9	13	16	18	22	29
16777216	16	4.0	15	5	13	22	7	10	13	15	18	23	29
16777216	16	4.3	16	6	15	31	0	8	13	16	19	23	31

A.3 Simulation Code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <limits.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>

#include <algorithm>
#include <string>
#include <vector>

using std::sort;
using std::string;
using std::vector;

struct RowInfo;

struct Id {
  unsigned d;
  unsigned operator[] (unsigned i) const {
    unsigned shift = (8 - (i+1))*4;
    return (d >> shift) & 0xF;
  }
  operator unsigned () const {return d;}
  Id() {}
  Id(unsigned d0) : d(d0) {}
};

static inline Id operator^ (Id x, Id y)
{
  return Id(x.d ^ y.d);
}

struct Node {
  Id id;
  void add(Id);
  int tally[8][16];
  void init(Id);
  void get_row_info(RowInfo &);
};
```

```
struct Network {
  char filename[5+8+1+2+4+1];
  unsigned network_size;
  unsigned num_samples;
  Node * samp;
  // the first num_samples added will also be added to samp
  void add(Id id);
  void init(unsigned num_samples);
  void write();
  void read();
  void clear() {network_size = 0; num_samples = 0; delete[] samp; samp = 0;}
  Network() : samp(0) {}
  ~Network() {delete[] samp;}
};

void Network::init(unsigned num_samples0) {
  delete[] samp;
  network_size = 0;
  num_samples = num_samples0;
  samp = new Node[num_samples];
};

void Network::add(Id id)
{
  if (network_size >= num_samples) {
    for (unsigned i = 0; i < num_samples; ++i)
      samp[i].add(id);
  } else {
    for (unsigned i = 0; i < network_size; ++i)
      samp[i].add(id);
    samp[network_size].init(id);
    for (unsigned i = 0; i < network_size; ++i)
      samp[network_size].add(samp[i].id);
  }
  ++network_size;
}

void Network::write() {
  int seq_num = 0;
loop:
  int res = snprintf(filename, sizeof(filename), "samp/%08u-%u.dat",
                     network_size, seq_num);
  assert (res < sizeof(filename));
  if (access(filename, F_OK) == 0) {++seq_num; goto loop;}
  FILE * f = fopen(filename, "wb");

  fwrite(&network_size, sizeof(network_size), 1, f);
```

```
  fwrite(&num_samples, sizeof(num_samples), 1, f);
  fwrite(samp, sizeof(Node), num_samples, f);
  fclose(f);
}

void Network::read() {
  assert(filename[0]);
  FILE * f = fopen(filename, "rb");
  fread(&network_size, sizeof(network_size), 1, f);
  fread(&num_samples, sizeof(num_samples), 1, f);
  delete[] samp;
  samp = new Node[num_samples];
  fread(samp, sizeof(Node), num_samples, f);
  fclose(f);
}

template <typename F>
void stats(unsigned n, const F & f, FILE * o)
{
  double total = 0;
  double total2 = 0;
  unsigned sizes[n];
  for (unsigned i = 0; i < n; ++i)
  {
    unsigned s = f(i);
    sizes[i] = s;
    total += s;
    total2 += s*s;
  }
  sort(sizes, sizes + n);
  fprintf(o,
          "%3.0f %4.1f | %3u %3u %3u %3u  | %3u %3u %3u %3u %3u %3u %3u\n",
          total/n,
          sqrt((total2 - total*total/n)/(n - 1)),
          sizes[static_cast<int>(n*0.50)],
          sizes[static_cast<int>(n*0.75)] - sizes[static_cast<int>(n*0.25)],
          sizes[static_cast<int>(n*0.95)] - sizes[static_cast<int>(n*0.05)],
          sizes[n-1] - sizes[0],
          sizes[0],
          sizes[static_cast<int>(n*0.05)],
          sizes[static_cast<int>(n*0.25)],
          sizes[static_cast<int>(n*0.50)],
          sizes[static_cast<int>(n*0.75)],
          sizes[static_cast<int>(n*0.95)],
          sizes[n-1]);
}
```

```
template <typename F>
void dump(unsigned n, const F & f, const char * fn0, const char * suf)
{
  char fn[strlen(fn0) - 4 + strlen(suf) + 1];
  snprintf(fn, sizeof(fn), "%.*s%s", strlen(fn0) - 4, fn0, suf);
  FILE * o = fopen(fn, "w");
  for (unsigned i = 0; i < n; ++i)
  {
    unsigned s = f(i);
    fprintf(o, "%u\n", s);
  }
  fclose(o);
}

struct RowInfo
{
  struct {int lsize; int fsize; bool complete;} d[8];
  int size;
  int leaf_size;
  int fsize;
};

struct GetSize {
  const RowInfo * ri;
  GetSize(const RowInfo * r) : ri(r) {}
  unsigned operator() (unsigned i) const {return ri[i].size;}
};

struct GetFSize {
  const RowInfo * ri;
  GetFSize(const RowInfo * r) : ri(r) {}
  unsigned operator() (unsigned i) const {return ri[i].fsize;}
};

struct GetLeafSize {
  const RowInfo * ri;
  GetLeafSize(const RowInfo * r) : ri(r) {}
  unsigned operator() (unsigned i) const {return ri[i].leaf_size;}
};

void Node::init(Id id0)
{
  id = id0;
  for (int row = 0; row < 8; ++row)
    for (int col = 0; col < 16; ++col)
      tally[row][col] = 0;
}
```

```
void Node::add(Id to_add)
{
  Id x = id ^ to_add;
  if      (x[0]) tally[0][to_add[0]]++;
  else if (x[1]) tally[1][to_add[1]]++;
  else if (x[2]) tally[2][to_add[2]]++;
  else if (x[3]) tally[3][to_add[3]]++;
  else if (x[4]) tally[4][to_add[4]]++;
  else if (x[5]) tally[5][to_add[5]]++;
  else if (x[6]) tally[6][to_add[6]]++;
  else if (x[7]) tally[7][to_add[7]]++;
}

void Node::get_row_info(RowInfo & ri)
{
  ri.size = 1;
  ri.fsize = 1;
  ri.leaf_size = 0;
  bool in_leaf = false;
  for (int row = 0; row < 8; ++row)
  {
    int filled_cols = 0;
    ri.d[row].lsize = 0;
    ri.d[row].fsize = 0;
    for (int col = 0; col < 16; ++col) {
      if (tally[row][col] > 0) filled_cols++;
      ri.d[row].lsize += tally[row][col];
      ri.d[row].fsize += tally[row][col] < 2 ? tally[row][col] : 2;
    }
    ri.d[row].complete = filled_cols == 15;
    ri.size += ri.d[row].complete ? ri.d[row].fsize : ri.d[row].lsize;
    ri.fsize += ri.d[row].fsize;
    in_leaf = in_leaf || !ri.d[row].complete;
    if (in_leaf) ri.leaf_size += ri.d[row].lsize;
  }
}

void print(const Node & n, const RowInfo & ri)
{
  printf("\n");
  printf("Node: %x\n\n", (unsigned)n.id);
  for (int row = 0; row < 8; ++row)
  {
    printf("  ");
    for (unsigned col = 0; col < 16; ++col)
    {
```

```
      if (col == n.id[row]) printf("_ ");
      else if (n.tally[row][col] >= 2) printf("2 ");
      else if (n.tally[row][col] == 1) printf("1 ");
      else printf(". ");
    }
    if (ri.d[row].complete) printf(" C");
    printf("\n");
  }
  printf("\nLeaf Size: %d\n\n", ri.leaf_size);
}

static inline Id rand_id()
{
  assert(INT_MAX == RAND_MAX);
  unsigned res = rand();
  return Id(res << 1);
}

void proc(int network_size, int num_samples)
{
  if (num_samples > network_size) num_samples = network_size;

  Network nw;
  nw.init(num_samples);

  for (int i = 0; i < network_size; ++i)
  {
    nw.add(rand_id());
  }

  nw.write();
  printf("%d size\n", network_size);
}

void analyze()
{
  Network nw;

  DIR * dp;
  struct dirent * ep;
  dp = opendir("samp/");

  FILE * o_size = fopen("nodes-size.txt", "w");
  FILE * o_fsize = fopen("nodes-fsize.txt", "w");
  FILE * o_lsize = fopen("nodes-lsize.txt", "w");

  vector<string> data_files;
```

```
  while (ep = readdir(dp), ep) {
    unsigned s = strlen(ep->d_name);
    if (s <= 4 || strncmp(ep->d_name + s - 4, ".dat", 4) != 0) continue;
    data_files.push_back(ep->d_name);
  }

  sort(data_files.begin(), data_files.end());

  vector<string>::const_iterator i = data_files.begin(), end = data_files.end();
  for (; i != end; ++i) {
    nw.clear();
    snprintf(nw.filename, sizeof(nw.filename), "samp/%s", i->c_str());
    nw.read();

    RowInfo ri[nw.num_samples];

    for (int i = 0; i < nw.num_samples; ++i)
    {
      nw.samp[i].get_row_info(ri[i]);
    }

    fprintf(o_size, "%8d | ", nw.network_size);
    stats(nw.num_samples, GetSize(ri), o_size);
    dump(nw.num_samples, GetSize(ri), nw.filename, "-size.txt");

    fprintf(o_fsize, "%8d | ", nw.network_size);
    stats(nw.num_samples, GetFSize(ri), o_fsize);
    dump(nw.num_samples, GetFSize(ri), nw.filename, "-fsize.txt");

    fprintf(o_lsize, "%8d | ", nw.network_size);
    stats(nw.num_samples, GetLeafSize(ri), o_lsize);
    dump(nw.num_samples, GetLeafSize(ri), nw.filename, "-lsize.txt");

  }
  closedir(dp);
  fclose(o_size);
  fclose(o_fsize);
  fclose(o_lsize);
}


unsigned rand_exp(unsigned min0, unsigned max0) {
  double min = log((double)min0);
  double max = log((double)max0);
  double r = rand();
  r = r * (max - min)/RAND_MAX + min;
```

```
  r = exp(r);
  return static_cast<unsigned>(r);
}


int main(int argc, const char *argv[])
{
  srand(time(0));

  if (argc == 2 &&  strcmp(argv[1], "proc") == 0) {

    unsigned max = 24;
    for (unsigned p = 8; p <= max; p += 2) {
      proc(1 << p, 256);
      fflush(stdout);
      proc(1 << p, 256);
      fflush(stdout);
      proc(1 << p, 256);
      fflush(stdout);
    }
    for (unsigned i = 0; i < 256; ++i) {
      unsigned r = rand_exp(64,1 << max);
      proc(r, 256);
      fflush(stdout);
      proc(r, 256);
      fflush(stdout);
    }
    return 0;

  } else if (argc == 2 &&  strcmp(argv[1], "analyze") == 0) {


    analyze();
    return 0;

  } else {

    fprintf(stderr, "Usage: %s proc|analyze\n", argv[0]);
    return 1;

  }

}
```

Appendix B

Implementation Details

An implementation for DistribNet is available at `http://distribnet.sourceforge.net/`.

B.1 Physical Storage

Blocks are currently stored in one of three ways:

1. block smaller than a fixed threshold (currently 1k) are stored using Berkeley DB (version 3.3 or better).

2. blocks larger than the threshold are stored as files. The primary reason for doing this is to avoid limiting the size of data store by the maximum size of a file which is often 2 or 4 GB on most 32-bit systems.

3. blocks are not stored at all, instead they are linked to an external file outside of the data store much like a symbolic link links to a file outside of the current directory. However since blocks often represent only part of the file the offset is also stored as

part of the link. These links are stored in the same database that small blocks are stored in. Since the external file can easily be changed by the user, the SHA-1 hashes will be recomputed when the file modification data changes. If the SHA-1 hash of the block differs all the links to the file will be thrown out and the file will be relinked. (This part is not implemented yet).

Most of the code for the data keys can be found in data_key.cpp

B.2 Determining the amount of space used

When determining the amount of space used only large blocks are considered. That is, only blocks which are stored as actual files will be counted. This is because predicting the amount of space a key will take to store in the database is not straightforward. It is easy to find out the current space used by a database file but it is not easy to determine what to do to decrease the size due to the large amount of meta-data stored in a database. With large blocks it is fairly safe to assume that the amount of space used is approximately the size of the file and that the size of the metadata is relatively insignificant. Thus to free a certain amount of space N, simply keep deleting files until the sum of the sizes of the deleted files is larger than N.

Of course, the amount of size the database used is significant and the amount of data stored in it should be limited. I am just not sure how to do that. The best solution may be to simply limit the number of entries stored in the database.

B.3 Language

DistribNet is written in fairly modern C++. It uses several external libraries however it will not use any C++ specific libraries. In particular I have no plan to use any sort of Abstraction library for POSIX functionally. Instead, thin wrapper classes are used which I have complete control over and will serve mainly to make the process of using POSIX functions less tedious, rather than abstract away the details of using them.

Bibliography

[1] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, Brandon Wiley. Protecting Free Expression Online with Freenet. IEEE Internet Computing. Volume 6, Issue 1, pages 40-49. January 2003.

[2] B. Cohen. Incentives build robustness in BitTorrent. In Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.

[3] Emulab. `http://www.emulab.net/`

[4] FastTrack. Wikipedia article. `http://en.wikipedia.org/wiki/FastTrack`.

[5] Michael J. Freedman and David Mazières. Sloppy Hashing and Self-Organizing Clusters. In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS), Berkeley, CA, February 2003.

[6] GNUNet. `http://www.ovmj.org/GNUnet/`

[7] The Gnutella protocol specification v 0.4. 2000. `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`.

[8] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, I. Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. SIGCOMM'03, August 25-29, 2003, Karlsruhe, Germany.

[9] David R. Karger, Matthias Ruhl. Finding Nearest neighbors in Growth-restricted Metrics. In Proceedings of the ACM Symposium on Theory of Computing (STOC), May 19-21, 2002, Montreal, Quebec, Canada.

[10] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In Proceedings of the ACM Symposium on Theory of Computing (STOC), 2000.

[11] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable Dynamic Emulation of the Buttery. In Proceedings of the PODC, 2002.

[12] Petar Maymounkov and David Maziéres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, March 2002.

[13] Petar Maymounkov and David Maziéres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. Lecture Notes in Computer Science, Volume 2429. pp. 53 - 65. P. Druschel, F. Kaashoek, A. Rowstron (Eds.)

[14] The Napster protocol specification. Last updated 2001. `http://opennap.sourceforge.net/napster.txt`.

[15] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA), June 1997.

[16] Sylvia Ratnasamy. A Scalable Content-Addressable Network. PhD thesis, University of California, Berkeley, October 2002.

[17] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of the ACM SIGCOMM 2001 Technical Conference, San Diego, CA, USA, August 2001.

[18] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In IFIP/AC International Conference on Distributed Systems Platforms (Middleware), pages 329-350. November 2001.

[19] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. IEEE/ACM Transactions on Networking (TON), Volume 11, Issue 1, pages 17-32. February 2003.

[20] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookp Service for Internet Applications. In Proceedings of the ACM SIGCOMM 2001, San Diego, CA, USA, Auguest 2001.

[21] B.Y. Zhao, K.D. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley, April 2001.