# A Friendly Peer-to-peer Network

Yongrim Rhee
Master's Thesis
ICT/ECS,
Kungliga Tekniska Högskolan,
Stockholm, Sweden

Supervisor: Erik Aurell
Examiner: Mihhail Matskin

15 June 2006

## Abstract

Peer-to-peer systems are typically designed to be deployed in an open environment at a global scale. Such deployment must ensure that peer cooperate and contribute. Without cooperation and contribution of peers, the quality of service of the overall system degrades.

From economics point of view, free riders exist when resources consumed by peers are public and cost free. Incentives algorithms in peer-to-peer systems attempt to reduce free rider problem over the public domain. Incentives, however, already exist in social groups in society. In Peer-to-peer groups formed by such social groups, people are more likely to contribute their computing resources and exhibit altruistic behavior. Furthermore, free riders are less likely to exists in smaller scale peer-to-peer networks. Smaller but many instances of peer-to-peer networks can help form peer groups from existing social networks. To this end, we implement a way of creating and controlling privatized peer-to-peer overlay network using the DKS structured overlay network.

Using the privatized DKS, we build a decentralized peer-to-peer ftp server called Fortress FTP.

# Contents

# List of Figures

# Acknowledgments

The work for this master's thesis was conducted at SICS (Swedish Institute of Computer Science) as a partial requirement for the master's degree at Kungliga Tekniska Högskolan.

Firstly, I sincerely thank Professor Erik Aurell for his relentless support and sound advice. His encouragements and experience were instrumental in scheduling and completing the project on time. Sameh El-Ansary has been indispensable in lending me his critical mind. Sameh's tenacity in solving problems was a source of inspiration. I especially would like to thank Per Brand and SICS for providing me with the equipment and space conducive to completing the project. The completion of this thesis would not have been possible without the expertise and the hands of experience of the DKS system of Antonis Thodis and Birgir Stefansson.

Finally, I thank mom for being a sponsor and an inspiration in life.

# Chapter 1

# Introduction

The global interconnectivity between computers provided by the Internet brought about the development of many distributed applications and triggered the interest and research in distributed systems by the scientific community. According to Hobbes' Internet Time line [26], the number of computers with a registered IP address was well over 300 million in 2005, and the number continues to grow. With each personal computer being a potential server in a peer-to-peer system combined with the ever increasing hardware performance, the Internet today provides a unique computing environment where peer-to-peer applications can be deployed.

An introduction to a P2P topic cannot be complete without the perhaps hackneyed reference to Napster[22]. Arguably, Napster popularized the P2P concept. Its server-client model allowed clients to find and share music files stored on other clients by storing search meta-data on its server. As it is with all the server-client model, the server in the Napster model was a single-point of failure and a point of bottle neck. Hence the Napster model did not scale well. A number of other distributed applications have appeared since Napster. Among them, file sharing applications remain especially popular. However, many of these earlier file sharing applications, though distributed, did not take into consideration the critical properties that define today's P2P systems such as scalability, robustness, and fault-tolerance, and thus was limited in many aspects of their designs. In addressing these issues, a number of peer-to-peer systems have emerged with more robustness in their designs.

Among the peer-to-peer infrastructures that have appeared recently, distributed hash tables (DHT) in particular have emerged as a well structured decentralized P2P data structure. DHTs allow correct routing of a key based lookup message with efficient lookup cost guarantees. DHTs can be the ground laying infrastructure for more complex applications. One of such applications is data storage. Several designs and implementations of data storage using distributed hash tables already exist, namely CFS [8], OceanStore [18], and PAST [10].

Many of the existing peer-to-peer systems ambitiously attempt to create systems at a global scale. In such systems, a huge number peers could potentially join the system. Despite the large numbers that appeals to scalable peer-to-peer systems in open networks, incentives must exist in such systems in order for peers to remain in the system and contribute to the system. Without such incentives, self interested users only consume resources without contributing back to the system. Such was the case with Della - it was discovered that nearly 70% of Gnutella users did not share files and over 50% of the search results came from 1% of the peers [1]. It is thus evident that level reciprocation in an open environment can be expected to be nothing short of pitiful without incentives.

Incentive mechanisms often model our own complex social models. Reputation based incentive mechansims proposed in [16] closely mimick reputation system in real life. TFT (Tit-for-tat) algorithm in use by the BitTorrent protocol [7] closely resembles how people tend to reciprocate to those who contribute in the first place. Barry in [4] even suggests that computer networks is a technology that affords us to extend our social networks.

Perhaps inspired by the early success of file sharing applications, peer-to-peer systems have been most often been envisioned as a global and public system. But just as people are less likely to help strangers than their friends, such peer-to-peer systems exhibit low cooperation and contribution from peers. Free riding peers plagued the earlier P2P systems and new proposals introduce incentive techniques to reduce or eliminate free riding in a P2P system[12]. On the other hand, if peers can be chosen by the administrators so that peers are known to exhibit positive cooperative behaviors before hand, the P2P system will have much higher reciprocation rate than publicly accessible systems without the use of incentive mechanisms.

People within a same social network have strong incentives to cooperate with each other; if members of a well reciprocating social group can form a peer-to-peer network with altruistic intentions, then it would obviate the need to build incentive mechanisms for its peer-to-peer network. In fact, having incentive mechanisms may hinder their altruistic preferences.

Private DKS extends the functionality of the Distributed K-ary System (DKS) [? ] by securing communications between peers and requiring authentication to join the system. This leaves the administration of peer membership up in the control of the users. Fortress FTP uses the Private DKS to build a distributed FTP server that is robust, peer-to-peer, scalable, and fault-tolerant.

## 1.1 Contributions

Peer join authentication has been implemented in 900 lines of Java code. The authentication protocol is added to existing code of the DKS System. The authentication protocol makes use of the public key infrastructure.

Fortress FTP has been implemented by modifying the Apache FTP Server and creating a distributed storage which the server accesses to it. The modified code adds about 3000 lines of code. Fortress FTP's advantages over other distributed FTP servers are that it provides load balance, fault tolerance and self-management in a robust and scalable way.

## 1.2 Thesis Overview

Related work section summarizes the peer-to-peer systems that have been most influential in writing this thesis. Section 3 is a review of the DKS, a distributed hash table developed at SICS. Section 4 explains how the DKS system is secured and privatized. Section 5 discusses how the privatized peer-to-peer network can be applied. The private group location service using the DKS is discussed in section 6. Section 7 overviews the architecture and implementation of the Fortress FTP in detail. We conclude the paper with in section 9.

# Chapter 2

# Related work

## 2.1 Peer To Peer

Peer-to-peer distributed systems remove the notion of client and server by allowing nodes to act as both server and client. This distributes the service functionality over the participating peers instead of placing the burden of service on a limited number of servers. Although not a panacea to network computing, peer-to-peer techniques have seen various successful deployments over the recent years, most notably in areas of file sharing and IP telephony.

Besides the removal of centralized control and failure, scalability is a property that characterizes peer to peer systems. As the number of participating nodes increase in a system, so does the resources that can be provided by the nodes. Efficient management and utilization of resources that scale well as the number of participating peers increase is an important goal in peer-to-peer systems.

Removal of the centralized server means that the role of administration must also be distributed. Because it would be impractical to manage every single peer by hand, peer-to-peer infrastructure must self-organize and mange the peers as they join and leave, as well as tolerate failures and malicious behavior.

Finally, a peer-to-peer system must take into consideration that the nodes in the system will most likely differ in their network capacity, computational power, and memory and storage capability.

## 2.2 Deployed P2P Systems

In the late 90s, Napster [22] file sharing service provided its users the ability to share files through its indexing server. Although file transferred took place directly between peers, the centralized indexing server was responsible for locating other peers and search queries. The centralized server is often cited as a point of failure and bottleneck. Napster model is hence not considered a robust nor scalable.

Gnutella [15] builds an ad hoc network where search queries can be flooded. It certainly has its advantages over the Napster's centralized model in terms of scalability and robustness. Due to the ad-hoc overlay network the Gnutella protocol builds, searching was not comprehesive and inefficient.

Peer-to-peer systems continues to emerge as a prevailent technology especially in Internet file sharing. Other applications for peer-to-peer technologies are beginning to appear. Even so, most peer-to-peer technologies revolve around data storage. Overnet is an example of a completely decentralized peer-to-peer system based on a distributed hash table (DHT) implementation called Kademlia [20]. Detailed discussion of DHT follows in section 3. Current generation P2P networks such as FreeNet [6] augment P2P features by provide anonymity.

## 2.3 Distributed Hash Tables

Distributed hash table is a peer-to-peer overlay network that efficiently routes messages to a peer given the peer identifier. Unlike existing ad-hoc overlay networks such as the aforementioned Gnutella protocol, the overlay network created by a distributed hash table is said to be *structured* because it provides guaranteed delivery of messages to the correct peer in the system while the number of hops required to deliver the message scales roughly logarithmic to the number of peers in the system. The dynamic environment that is characteristic of a peer-to-peer system requires that the DHT overlay networks be fault-tolerant and self managing as peers leave, join and even fail.

## 2.4 Distributed Storage using DHT

Distributed storage is one of the well explored idea built on top of the distributed hash table concept. The desirable properties upon which DHT is built makes it an ideal substrate to build peer-to-peer storage applications on top of. PAST [10], CFS [8], and OceanStore [24] implement distributed file systems using Chord, Tapestry, and Pastry, respectively. All tree implementations seek to create persistent wide area data storage that are self organizing and self managed.

Cryptographic hashing is a widely used in implementing distributed storage systems. Cryptographic hashes are quasi-random and helps in evenly distributing the hash values within the hash value space. This feature is especially attractive to distributed systems because key assignments to data objects can be somewhat evenly distributed when they are generated by hashing their content.

### 2.4.1 PAST

PAST generates a 160-bit file Id and stores the file at a node which matches the most significant bits of the file Id to the node's 128-bit identifier. Both folds and node Ids are generated using cryptographic hash of the file and nodes meta information. This helps in probabilistically increasing diversity of locality and capability as well as uniformly distributing the nodes in the identifier space. Replication factor $k$ can be specified for a file and the replicas are stored at $k$ nodes that have the closest matching id to the most significant 128-bits of the 160-bit file Id. Again, because of the uniform distribution of the nodes, the replicas are just as well replicated over $k$ diverse nodes.

PAST's security model provides users with anonymity, privacy of data and enforces quotas. The model revolves around the use of smart cards. Anonymity is provided by using the smart cards signature as a user's pseudonym. The smart card is also used to create node Ids by cryptographically hashing the public key contained in the smart card. The insert operation verifies the integrity of the data by returning a certificate that contains the secure content hash of the file. Quota information is maintained on the smart card to prevent users from exceeding their given quota.

### 2.4.2 CFS

CFS [8] is a decentralized file system developed using a layered approach. Similar to the file system semantics deployed in UNIX, CFS creates file system structure at a data block level. A data block storage layer, called Dhash, is built over the Chord routing layer to provide data block storage and retrieval using identifiers to data blocks. The block ID is determined by cryptographically hashing the block's content. This allows the content of the data block to be verified using the block's ID. Another benefit is that it probabilistically spreads the blocks among the key space thereby providing a good distribution of the data blocks among the peers. Dhash data blocks are typically in tens of kilobytes.

The CFS file system layer is built over Dhash. Except for the root block, entire file system structure is created using directory blocks, inode blocks and data blocks all of which are inserted and retrieved using Dhash using simple put and get interfaces. A root block is inserted using the public key as its identifier, and is signed with the public key to enable verification of the root block.

CFS is intended to provide a distributed file system that scales well while providing performance that is competitive with the current point-to-point transfer protocols. Having small block size (tens of kilobytes) and distributing blocks to many peers balances the load among the peers. Block are cached at nodes that are on the lookup path of the block to provide additional load balancing. Network latency for a block lookup is masked by fetching multiple blocks concurrently.

### 2.4.3 OceanStore

OceanStore's design aims to provide highly durable storage in a self managing and self organizing system that is resilient to fault and failures. Byzantine agreement protocol, erasure encoding and Tapestry routing scheme is used to satisfy these requirements.

Storage unit in OceanStore is captured by the notion of a data object. Data object is composed of data blocks. Much like CFS, data blocks are identified by the cryptographic hash value of their content. A tree structure formed by storing the identifiers to data blocks in data blocks themselves create a version of a data object. Because data objects are cryptographic hashes of their contents, data object is self verifying. Any change to a data object require a new root block and therefore a root block represents a version a data object. Because the hashing is consistent, data blocks that remain unchanged in new versions are still referenced by the newer root blocks. Every version of the root block is kept by the system to allow play back of changes and is assigned a single identifier.

All data blocks are archived in the OceanStore model to provide long term storage. To increase durability of data objects, data blocks are stored as erasure encoded fragments. Erasure encoding increases durability by increasing the number of fragments while requiring only a fraction of those fragments to reconstruct the original data. For instance, a data block can be erasure encoded into 32 fragments while requiring any 8 pieces of the fragments are required to reconstruct the original data block. This yields the encoding rate, defined to be m/n (where m is the number of pieces required to construct the data and n is the total number of fragments) to be $(8/32)$. Storing erasure encoded fragments has its drawbacks of increasing the storage overhead by roughly the inverse of the encoding rate and that it is computationally expensive to reconstruct a data block.

A separate group of servers called the inner ring handles updates and changes to data object by creating a instance of the data object. This instance is called a primary replica. All updates to a data object is directly applied to the primary replica by the inner ring. Unlike CFS and PAST where most nodes are assumed to be well behaved, OceanStore makes a Byzantine assumption - i.e., no more than N/3 -1 nodes are assumed to be faulty. Byzantine agreement protocol takes place before changes are applied to the primary replica by the participating servers. If a primary replica does not exist it is constructed from the erasure encoded fragments. All changes to the primary replica to secondary replicas in a dissemination tree constructed by servers outside of the inner ring. Secondary replicas provide caching of a data object to applications.

## 2.5 Content Distribution

### 2.5.1 BitTorrentContent Distribution Protocol

BitTorrent [5] is a content distribution protocol where the role of transferring files are distributed among peers. Peers increase their chance to be chosen to be uploaded to by other peers by an incentive based algorithm. Cohen asserts that it is the incentive algorithm, called TFT (Tit-for-tat), that is responsible for promoting cooperation among peers in [7]. Each instance of the protocol, called a torrent, typically transfers a single file.

In order to begin a torrent session, a .torrent file is downloaded from a web server. A .torrent file contains the address of a server that keeps track of the peers participating in the torrent (hence called the tracker), along with meta information about the files. Included in the meta information is the SHA-1 hash values of the individual sections of the file so that the pieces can be verified as they are received.

When a BT client contacts a tracker to bootstrap itself to a torrent, a random set of peers of a set size (usually 50) is chosen from the peer list by the server and sent to a peer. Besides the periodical statistical updates by the peer, the tracker has no other role in a torrent session. Because each peer's peer set is received randomly, the peers in the set are inevitably members of other peers' peer set. From a global view, this intersecting set of peers create a swarm of peers.

BitTorrent peers execute algorithms according its local view of the swarm. From the initial set of peers received from the tracker, a peer initiates connections to 40 of them. 40 additional connections are allowed to be initiated by other peers. If the total number of peer set drops below a set number, it can contact the tracker to receive a new set of peers. When a connection is initiated, the peers exchange information on pieces each peer owns. This information is represented in a sequence of bits and is continually updated by connected peers. Having this sequence of bits for all its peers, a local peer can then deduce which pieces are rarest among the peer set and which peers among the peer set has pieces the local peer does not. Using this information, rarest pieces are downloaded first from peers who have the missing pieces.

To maximize the upload bandwidth, a local peer chooses 4 peers to send pieces to. Peers providing the fastest download rates are chosen. This tit-for-tat algorithm, called the choke algorithm, discourages free-riding and rewards peers that provide the best bandwidth. Every 30 seconds, however, a randomly chosen peer is selected to send pieces to. This algorithm, called optimistic unchoke, allows the local peer to probe for who may be able to provide a faster download rate. It also gives newly arriving peers opportunity to grab pieces - because a new peer needs to get pieces in order to start trading among other peers, until the first 4 pieces are received, it downloads any 4 pieces the first chance it gets. The choke algorithm and the optimistic unchoke algorithm are central to the BT protocol in efficiently utilizing the swarm's collective bandwidth.

# Chapter 3

# DKS, a DHT

## 3.1 Distributed Hash Tables

Distributed hash tables can be thought of as a hash table object that is shared among many servers. Each server is responsible for a numeric key range. All the servers key range combine to cover the entire predefined key range of a distributed hash table. The servers therefore must keep routing tables so that hash table operations `put(key, value)`, `get(key)`, `remove(key)` can be performed by the server responsible for the key and that the request is routed correctly and efficiently. This is referred to as *key based routing.* The key space is often visualized in a circle to represent its modular arithmetic.

### 3.1.1 Lookup

The lookup operation, or routing, is fundamental to distributed hash tables. The `put`, `get`, and `remove` operations all require lookup as a part of its operation. Each node keeps a routing table to a number of other nodes in the system. There must exist a balance between the size of the routing table and the number of nodes in the system for the overlay network to remain scalable. The lookup must remain correct and efficient. In other words, a network topology must carefully be designed with respect to scalability, correctness, and efficiency.

Imagine an overlay network where each node points to another node to form a ring topology. In such ring topology, each node would only need a routing table of size 1 but it would yield a linear search time. On the other hand, an overlay network in which every node keeps a routing table of every other node in the network (a.k.a *clique* would yield a *O(1)* lookup time whilst the size of routing table would be *O(N)* which would be much too prohibitive to maintain in a dynamic network environment. See Fig. 3.1.

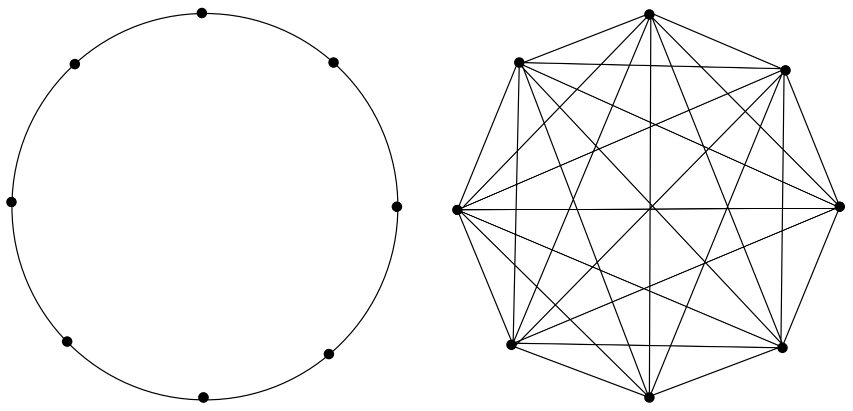
Figure 3.1: A ring and a clique topology

Different designs of distributed hash tables exist, for instance Chord, Pasty, Tapestry, and DKS. Although they deploy different lookup algorithms, the idea similar - lookup request is re-routed to nodes that are closer to the given key in geometrically decreasing jumps until the request arrives at the node that is responsible for the key range. Most routing algorithms resemble binary search and yields lookup cost in number of hops of $O($lg $n)$. It is with the routing tables that DHTs form a *structured* overlay network. Table 3.1 summarizes existing hash tables and their properties.

| | lookup cost | routing table size |
|---|---|---|
| DKS | $O(log_k\ N)$ | $O((k-1)log_k(N))$ |
| Chord | $O(\log(N))$ | $O(\log(N))$ |
| Pastry | $O(\log(N))$ | $O(\log(N))$ |
| Tapestry | $O(\log(N))$ | $O(\log(N))$ |

Table 3.1: DHT implementations and their properties.

### 3.1.2 Self Management

When nodes join a DHT system, the key space partitions are assigned automatically. During joins, the neighboring nodes update their routing table and hand over the data to the new node. Conversely, as nodes leave the system, the leaving node hands over its data items to the node. As nodes join and leave a DHT, routing tables must be updated to maintain correctness. Nodes joining and leaving peer-to-peer system is called *churn*. Self-healing of routing table under churn is an essential feature of DHTs.

### 3.1.3 Handling Failures and Data Resilience

To increase robustness, it is important that distributed hash tables handle node failures and resist data loss due to failures. Chord for example uses *periodic stabilization* where periodic messages are sent between nodes to detect failures. Data is often replicated over nodes to varying degrees in the distributed hash tables to increase resilience to data loss.

## 3.2 DKS

DKS (Distributed K-ary System) [19] is a distributed hash table. Unlike most existing DHTs, DKS aims to provide a generalized DHT infrastructure that is customizable to application specific needs. An instance of a DKS system is defined by *DKS(N,k,f)* where $k$ is the configurable search arity within the network and $N$ is the maximum number of nodes that can be in the overlay network. $f$ specifies the replication factor.

### 3.2.1 Lookup in DKS

Each node in the DKS system maintains $log_k N$ *levels*, and each level contains k consecutive *intervals*. Figure 3.2 represents intervals in 3 levels. The interval length $d$ at level $l$ is defined by:

$$d = N/(k^l)$$

For each level the intervals begin at the node's identifier. Pointers to the first node that lies within each interval is kept in a routing table. A lookup begins by locating the interval which the key lies and the node referenced by the corresponding pointer. If the node does not hold the data item, then the lookup is recursively called to node. This effectively creates a k-ary search tree. Due to the k-ary tree DKS constructs for its lookup, the lookup cost in DKS is $O(log_k N)$ while the routing table size is $O((k-1)logk(N))$.

### 3.2.2 Topology Maintenance

Maintaining routing tables as nodes join, leave and fail is essential to the robustness of an overlay network. In a DHT, because it is a structured overlay network, it is critical that the routing tables of nodes must ensure correctness.

Intuitively, when nodes join a DHT network, it must notify all nodes that should point to it. Nodes leaving the network informs all the nodes that are pointing to it that it should point to its *successor* Such graceful departures

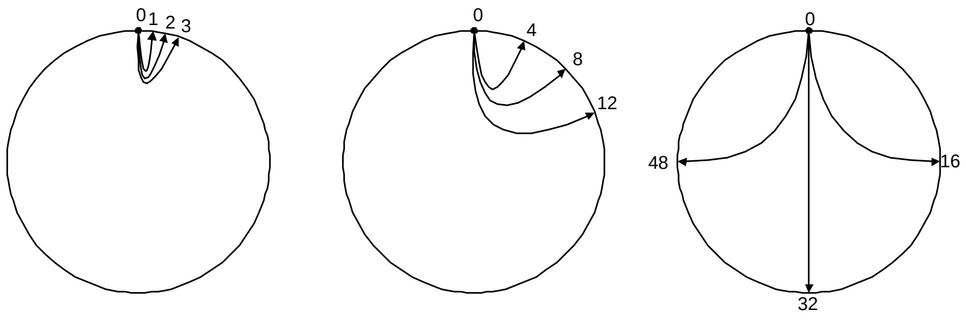

Figure 3.2: Intervals for each level maintained by DKS node 0 where $N = 64$ and $k = 4$.

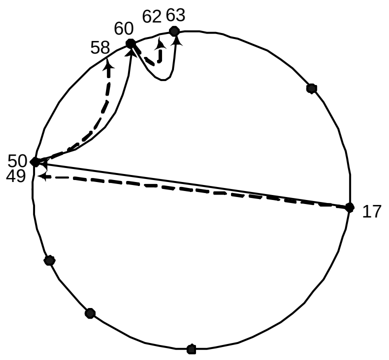

Figure 3.3: A lookup by node 17 for key 63 in DKS where $N = 64$ and $k = 4$. Solid dots indicate nodes. Dotted arrows indicate the original interval and the solid arrows indicate the actual interval.

allows nodes to self-organize in stepwise manner. Of course, node failures must also be taken into account when constructing a robust peer-to-peer system. Most DHTs use *periodic stabilization* to detect node failures where messages are periodically sent to nodes in the routing table to check if they have failed or not. A couple of issues to using this technique is that it steadily uses bandwidth and that the stabilization messages should be adjusted according to expected rate of churn - that is, if the rate at which stabilization messages are sent is high while the churn rate is low, then the stabilization messages can be wasteful whereas having too low of a rate would result in incorrect routing table entries.

DKS generalizes node failure events and node departure events into a single leave event through the use of a technique called *correction on change* [14]. When nodes either depart the system or fail, the affected nodes, that is, the nodes that have pointers to the said node, must be notified of the node so that the pointers on each of the affected node can be updated. Notification of those nodes can be done by the node that leaves, or by the node that detects the failed node. Using this technique, amount of stabilization messages sent by DKS depends solely on the level of dynamicism of the overlay network.

### 3.2.3 Replication

Most existing DHTs use successor-lists to store a node's replicated data at its successors. The rationale is that if the nodes are diversely distributed over the identifier space the DHT, then the probability that the node *and* all its successors will fail simultaneously is low. There are some inflexibility to this method. First, changes to the DHT membership means that successor lists needs to be updated by the affected nodes, and subsequently the replicas must also be redistributed accordingly. Another drawback to replicating at successive nodes is that replicated copies cannot be accessed as a means to balance the load without contacting the node first. Lastly, although a less relevant issue to F2F networks, assigning a single node as an authority to the respective data object allows the node to carry out a mendacity attack.

Instead of storing replicas at successive nodes, DKS stores them at a number of other nodes whose lookup operation can be independent of each other. This is made possible by assigning every node the responsibility of ranges that are equivalent classes of its own. This type of replication is called *symmetric replication* [2]. Specifically, for a replication factor *f*, and the replicated copy $x$, an equivalence class of $x$ is defined as: $r(i, x) = i + (x - 1)N/f$.

# Chapter 4

# Privatizing DKS

Security is a broad topic that needs to be put into context before further discussion. In this thesis, we're primarily concerned with security of network connections between nodes that form a private network using DKS. Network security can be achieved using a number of existing methods. For instance, using traditional methods, nodes can be on an already existing private network or use technologies such as VPN (Virtual Private Networks) to secure each connections among peers. Unfortunately, the availability of private networks is usually out of reach to most people. Furthermore, using point-to-point encryption such as VPN requires that all nodes in the system have the static verification information regarding each other and requires extensive knowledge and administration from the users. In order to secure the private overlay network formed by the DKS nodes in an open network (such as Internet), we use mainly asymmetric key exchange to authenticate membership of a DHT and data encryption to prevent eavesdropping by unauthorized users in the open network. The assumption to be made by privatizing the DKS node is that the nodes in the network is trusted. This assumption takes away the some of the security issues present in peer-to-peer networks formed by untrusted peers. Specifically, it augments data privacy and confidentiality to the system. Moreover, the threat of denial-of-service attacks from an unauthorized peer is reduced.

To provide a secure peer-to-peer network with data privacy, all communication between nodes in a DHT should be securely encrypted. Besides the secure communication links, additional security should be provided by requiring authentication by nodes joining the system. Verification of the peer using public key infrastructure enforces authorized-only joins to the DHT system. Thus only the node that can prove that it has the authority to be part of the system will be able to authorize new peers to the system. Two main components enforce security in a public DHT:

- **Secure communication between peers.** Messages and data among peers are encrypted.

- **Membership to the network excluded to non authorized hosts.** No node can participate as a peer in the DKS unless it can authenticate itself as an authorized member.

The above steps induce a secure peer-to-peer network where the security responsibility is left up to the users of the system.

## 4.1 Asymmetric Key Exchange

Public key infrastructure is used to authorize peer joins in the DKS. Key exchange takes place through out of band methods such as email or face-to-face exchange. In order to become a node in a DKS system, the node joining must first obtain the public key of the group it wants to join. We rely on the fact that in asymmetric key exchange, one's ability to decrypt a cipher encrypted with his public key authenticates his identity; only the authentic user will have his own private key therefore his ability to decrypt any cipher encrypted with his public key *must* mean that he is the authentic user. Each peer's identity will be verified using his public key using this concept. Similarly, with the group public key, the peer group can be identified by the peer when joining.

A table of authorized peer should be present within the peer group. This table contains the key of the authorized peer and is mapped to its hash key. We use a collision resistant SHA1 hash of the key for the key value. The peer list can be stored on every peer in the peer group. However, this requires that each node store keys of every other member. Change in the list of keys means that the change must be pushed to every node in the group as well. We take advantage of the fact the peers that have already joined the group is part of the DKS lookup network and store the members' keys using the DKS system. This way the authorized peer list is distributed and does not require notification to every peer in the system after an update to the list.

## 4.2 Key Insertion

Public keys are stored in the distributed hash table DKS. Key associated to a public key is the cryptographic hash of the key's byte representation. Key insertion can be performed by any of nodes already in the private DKS. Although not implemented for this work, this key insertion can be further limited by the use of passwords or verification methods to limit control of membership to administrative users.

Once the public key is inserted in the DKS system, the `get()` operation can be used to retrieve the key. This means that the peers can check only for the existence of a certain public key without querying the entire key range of the DKS. Indeed peers do not have to know the key values for every peer because when joining the group, the new node will transmit the hash value of its own key. The bootstrapping node can then lookup the key to begin the verification process of authenticating the node. Keys are inserted into the DKS system using the following operation:

```
addKey(PublicKey key)
```

## 4.3 Key Removal

Once the key is removed from the DHT, peers can no longer identify themselves using that key. Membership control can be actively or passively managed. The peer whose key has been removed from the system can be allowed to stay in the system. This can be useful in situations where machines should be allowed to be part of the system only once - the peers key can be removed as soon as it has joined. Actively managing peer participation means that the node identifying itself with key that have been removed from the system will be considered as faulty and will be dropped from the system. Keys from the system is removed using the normal remove operation of DKS:

```
removeKey(PublicKey key, boolean active)
```

## 4.4 Bootstrapping

In order to join a DKS instance, a peer must first obtain the address of the bootstrapping node. Finding the bootstrapping node can be done in two ways. First, it can be notified to the user through out of band communication. This can certainly happen at the same time when the out of band exchange of keys occur. The other method is to use a public directory service that provides a name to bootstrapping address, similar to DNS. In this case, the mapping in such systems would be the groups public key to bootstrapping information to the group. Acquiring bootstrapping information through a directory service is discussed in detail in chapter 6.

Public key must be inserted into the system before the node can join the system. The bootstrap process is sketched in the following steps:

1. Connect to the Server(bootstrap node).
2. Client(new node) sends the hash value of the its public key.
3. Server looks up the public key using the hash value using DKS and retrieves the public key.
4. Server generates challenge cipher and encrypts it using the public key.
5. Client receives the cipher and must decrypt it.
6. Client sends the decrypted challenge back to the server. (Client authentication complete)
7. Client generates the challenge cipher using the group public key.
8. Server receives the cipher and must decrypt the challenge. (Server authentication complete)
9. Client receives the group private key.

Once the above handshaking process steps are successfully taken, the node can complete the join process normally. The purpose of retrieving the group's public key is for other newly joining clients to verify the authenticity of the P2P group it is joining. If a client only received a symmetric key, although it can encrypt communication from that point it would do so without authenticating the group. Client(new node) authentication pseudo-code is in figure 4.2 and Server(peer node) authentication pseudo-code is in figure 4.1.

```
n.authenticate()
  n'=connection.accept()
  key = dks.getKey(n'.getKeyHash())
  challenge = generateChallenge(key) // generates challenge string
  cipher = encrypt(key, challenge) // encrypts it
  challenge' = n'.decrypt(cipher)
  if(challenge <> challenge')
    return false;
  endif
```

Figure 4.1: Pseudo-code for authentication for new peer.

```
n.authenticateGroup(node bsNode)
  challenge = generateChallenge(groupKey)
  cipher = encrypt(groupkey, challenge)
  challenge' = bsNode.decrypt(cipher)
  if(challenge <> challenge')
    return false;
  endif
```

Figure 4.2: Pseudo-code for group authentication.

## 4.5 Trust

We use the asymmetric key exchange to authenticate users and enforce trust within the peer group. The term trust as we have used it so far means trust among peers in the group. The classical issue of trusting the key, i.e., the authenticity of the key's owner, is considered to be solved and is out of the scope of this paper. Whether the key is authenticated using a certificate authority or web of trust, authenticating keys can be implemented at the application level and hence is not further discussed.

Enabling group level trust allows peer groups to form in a secure way. Privatization of peer-to-peer group allows the DKS overlay network to form a friendly network where contribution and cooperation among peers are expected to be much higher than the peers in an open and public peer-to-peer systems.

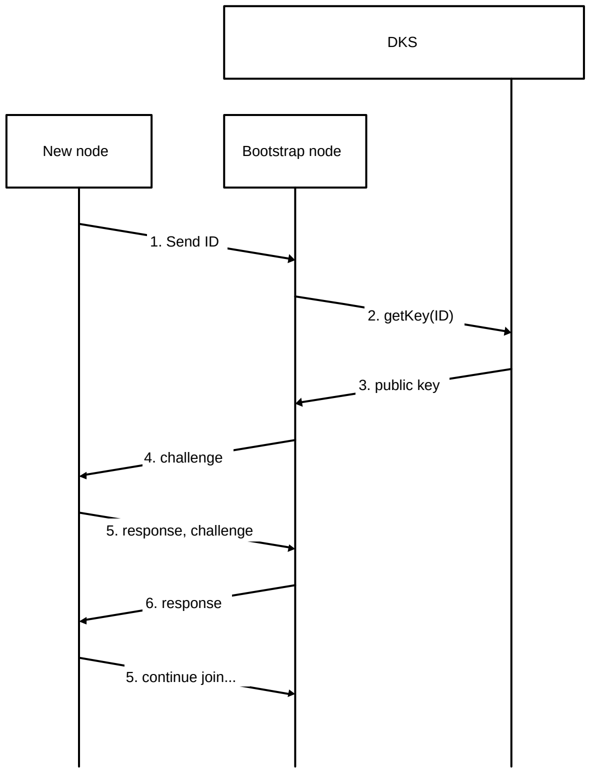

Figure 4.3: Authentication protocol between the joining node and the bootstrap node.

# Chapter 5

# Friendly P2P Networks

With the ever increasing processing power and storage space of the personal computers, peer-to-peer solutions are gaining popularity in both practice and research. Most peer-to-peer systems recognize the potential gains by enabling cooperation between computers across the Internet. However, peers rarely contribute or cooperate unless given the incentive to do so. This is because there is a person behind every peer and a person is no more likely to help a stranger than allow his computing resources to be shared to unknown group of people. Furthermore, the free rider problem will always exists in a system where payment or incentive system is not present. Private and smaller scale peer-to-peer networks reduce the number of free rider problem and are expected to be more cooperative. The private peer-to-peer group is expected to be formed by members of existing social groups. The members of the group is expected to contribute their computing resources, mainly i) bandwidth ii) storage space iii) CPU cycles and iv) shared data. We call this type of network *friendly* P2P network.

## 5.1 Background and motivation

Dunbar in [11] suggests that the typical size of social networks in human is approximately 150 due to the size of neocortex in human brains [11]. On the other hand, studies in economics suggest that this size limit is due to the increasing difficulty a social group will face in tracking free riders with larger group size. Free rider problem is more prevalent over public resources with large number of peers accessing it. We expect less frequency of free riders and higher level of contribution by privatizing the network and thereby privatizing the data contained within the overlay network.

### 5.1.1 F2F networks

F2F (Friend-to-friend) networks has been proposed in [25]. WASTE builds an ad-hoc mesh type network similar to Gnutella network [15] by allowing connections from trusted peers only. Nodes in WASTE achieve anonymity by redirecting data to trusted peers only. Because communication only takes place between trusted users, the WASTE network is not expected grow beyond 50 nodes.

In creating a privatized peer-to-peer network, peering relationship between real persons can extend to the peering relation of their machines. Unlike WASTE, we broaden the trust issue to the group level so that the peer-to-peer network can scale and reap the benefits of a structured overlay network of a distributed hash table. Peer-to-peer network within a social group where incentives to contribute and cooperate preexist can be expected to provide higher level of performance while providing the same benefits of a structured overlay network.

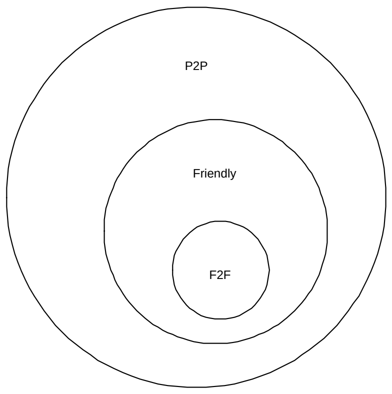

Figure 5.1: Relationship between P2P, F2F, Friendly networks.

Forming a purely F2F network is rather restrictive - peering relation *must* be with peers that are only trusted. In fact, WASTE network is not expected to grown beyond 50 nodes due to combined restriction of the restrictions set by the peering relation (strictly friend-friend), and the overhead of maintaining it. This is too restrictive in building a scalable P2P network. We propose a friendly network where trust is transitive; i.e., we trust friends of our friends, building a peering relation based on a web of trust.

Of course in real life, this is friendship, or trust, is not transitive in social networks. If that was the case, perhaps the world would be a much more trustworthy environment. Unfortunately it is not the case, and to make the matters even more complex, the degree of trust a person may place on another is a variable. The loosened requirement to form networks in this manner perhaps allows more dynamic formation of peer groups.

Hales from [17] has suggested that it is the formation of *tribes* that is attributed to BitTorrents success, instead of the widely credited TFT algorithm. The process in which friendly networks is formed by invitation of peers resembles the forming of tribes. Although not considered in this paper, the feature to monitor performance and contribution level of others can introduce a more explicit incentives for peers to be more cooperative without implementing them directly as a computed mechanism.

## 5.2 Benefits

In distributed systems, servers and clients must discourage, prevent, or be resilient to attacks through the use of security. Furthermore, in peer-to-peer systems, systems must encourage the *cooperation* between peers. Though cooperation can be encouraged and rewarded through the system design, such design requirements require complex and expensive system requirements while systems that are built upon existing social forces can relax those requirements.

Data resilience is often achieved by means of creating replicated copies over many nodes. Costs of replication is expected to be high in global systems where users often leave after insertion of their data for later retrieval. If the nodes in a system is expected to have much longer duration, then the costs of replication can be reduced to a significant amount.

In publicly available systems, protection and tolerance from faults or attack in the system is provided by the design criteria of the system. For example, systems based Byzantine agreement protocol must assume that no more than N/3 - 1 nodes in the system are faulty or malicious. Other systems limit the capability of each peer so that a single peer is not given enough authority to incur destructive actions, intentional or not, to the peer-to-peer system as a whole.

If the control of membership of peers in the peer-to-peer system is left to the users, then it is the users who must enforce and supervise the cooperation among peers in the system. This does not mean that the peer-to-peer system must be micro-managed. Specifically, if the frequency of the joins and leaves are reduced, and permanent departures from the peer-to-peer system is expected to be an infrequent and rare event, such system will require much less maintenance traffic as well as reducing the data replication factor. Li in [13] points out that DHTs in operation in Planet Labs have had stable operating conditions and almost never lost data during their normal operations.

Benefits provided by friendly peer-to-peer networks:

- Provide private groups
- Higher level of contribution and cooperation
- Lower instances of permanent departures
- Lower level of maintenance traffic

The performance of the overall peer-to-peer system is expected to be dependent on contribution. In a friendly peer-to-peer network, it is up to the users to screen possible users based on their social merits and to keep potential free riders from into the group. Deteriorating service quality of a friendly peer-to-peer network will probably result in peers to seek out better performing peer groups.

## 5.3 Enabling Friendly Network Concepts in DKS

One of the expected advantage of friendly peer-to-peer environment is that friendly networks are expected to reduce maintenance traffic significantly. In a data intensive peer-to-peer application, maintenance traffic and data replication costs should be minimal. It is unreasonable to expect peers to contribute large amounts of data in an open environment.

# Chapter 6

# Peer Group Location Service

Private peer-to-peer networks are not expected to be as large as open peer-to-peer systems. Bootstrapping information can be private and not widely published. We sketch a publicly available group location service that can be used by peers to retrieve information about a group they wish to join.

One of the issues when dealing with joins to peer-to-peer overlay network such as distributed hash tables is that because the system has many nodes, each node can potentially become a point of entry to the overlay network. Although it is possible to have a single static point of entry to a peer-to-peer system, failure of such designated entry point means that nodes can no longer able to join the system. For example, BitTorrent's tracker has often been criticized because it was the central point of entry for BitTorrent clients.

Early peer-to-peer systems recognized this problem and often published a list of well known points of entry into its overlay network so that clients can select a bootstrapping node to connect to. Such locally kept lists become outdated if not updated often because nodes in distributed peer-to-peer systems are not expected stay in the system permanently.

Several existing file sharing applications, namely [23] and [5], already use distributed hash table as a peer locating system. The continually updated list of peers are kept at a single location on a DHT. Although this is a sound solution, it has a couple of shortcomings. One of them is that it creates points of congestion for popular peer group lookup because the peer list resolve to a single key. This however, can be mitigated by the use of caching mechanism on the DHT itself. A possible solution to this is that a data structure holding such list can be distributed over the DHT and is the focus of on going research.

Although a peer can expect to receive addresses of other peers at the same time the keys are exchanged, IP addresses are often cumbersome for users to handle and often times ISPs do not provide static IP addresses to their customers. Peer group location service uses a DHT locate updated bootstrapping information to the private network to simplify the join process from the users perspective.

## 6.1 Design

We make the use of an open distributed hash table to implement the lookup service. Bootstrapping information is small in size - even a list of 1000 peer address would only be 4 kB. Having such a relatively low amount of data on the distributed hash table while probably having more than one peer to each list makes the lookup service relatively light weight and ideal to use a distributed hash table.

Using the lookup service, joins to a private network takes place in two steps. First, the bootstrapping information is obtained from the group location lookup service. The list is used then to bootstrap to nodes in the private ring. Invalid entries are corrected and the updated list is re-inserted back into the lookup service. This way, the list remains updated. The newly joining node can optionally insert itself into the list if it wants to announce itself as a possible bootstrap node in the private network.

Each network group has a public key associated with it. It is used for nodes joining to authenticate the group when joining. Another use it has is to name the group during the peer group location. The group's public key is used as a key during the lookup for the group. Updates to the bootstrapping information of the group is opaquely signed and can therefore be verified. Although this does not prevent malicious peers to insert false data associated with a group's key, the false data can be detected by checking the signature. Additionally, since the group lookup service is only used during the bootstrapping process, it is not critical to protecting data in the private network.

## 6.2 Implementation

DKS is used as the distributed hash table to implement the lookup service. It is implemented in Java. A single instance of DKS node to the group location service runs concurrently with the one or more instances of DKS nodes in the

```
getGroupInfo(groupKey);
updateGroupInfo(groupKey);
```

Table 6.1: Interfaces to the group location service.

private network. The idea is that a machine can have multiple instances of DKS nodes to different private networks it may belong to. During the join phase, the bootstrapping information to all the private networks a peer is part of are retrieved from the group location service using the group location service. Interfaces to the group location service is shown in Figure 6.1.

# Chapter 7

# Application: Fortress FTP

Fortress FTP is a peer-to-peer FTP repository. Fortress FTP does not serve any local data, but data is virtualized over the peer-to-peer overlay network. It is fault-tolerant; it probabilistically guarantees that a single or a fractional loss of peers will not affect the availability and the durability of the data. It is self-organizing; the peer-to-peer network reacts and organizes itself to peer joins and leaves. Furthermore, the peer balance the load automatically.

Fortress FTP is built on top of Private DKS as described in chapter 4. In order to join an existing Fortress FTP site, the new node's public key must have been added to the private DKS.

The FTP (File Transfer Protocol) has been around for almost as long as the Internet has been around. It is familiar to most Internet users and their browsers therefore it makes an ideal interface to expose a peer-to-peer storage system because only the providers of the service will have to be aware of peer-to-peer nature that is at work behind the servers. We build a scalable, peer-to-peer, data repository accessible via FTP. Having such legacy protocol exposed to users has multiple advantages:

- Fortress FTP can provide access to the data in the peer-to-peer network using existing FTP clients.
- Fortress FTP can be optionally configured to access data that is present only at the local machine also.
- FTP service does not have to provided at every node of the peer-to-peer system.
- The users of the FTP server does not have to be aware of the peer-to-peer system that is at work behind the scene.

- Existing FTP protocol interface allows existing applications to access the system.

An advantage that may not be immediately obvious is upload bandwidth can cumulate. This is particularly useful when peers are connected with an asynchronous bandwidth connection where up-link is significantly slower than the down-link. Many home subscribers to commercial Internet broadband service have this type of connection. A Fortress FTP peer will receive pieces of the files from many different other peers. If a file can be cooperatively downloaded from multiple peers, the limited upload capacity of a single peer will be masked. In Fortress FTP files are divided into chunks and multiple chunks are retrieved concurrently. The following sections describe in detail the design of Fortress FTP works.

## 7.1 Infrastructure

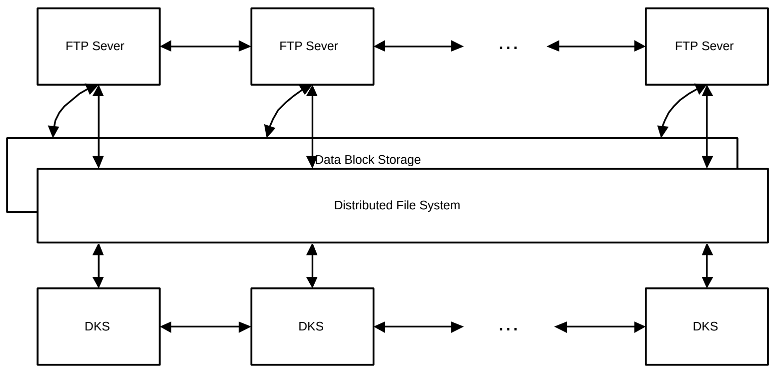

Figure 7.1: Structure of the Fortress FTP. The distributed file system is a virtualized resource created on top of DKS. Fortress FTP servers access the same virtual distributed file system. Data blocks are not stored in the distributed file system - only the directory structure is stored using DKS. Data blocks are exchanged among peers though the Data Block Storage module.

The underlying distributed file system in Fortress FTP is well suited for distributed peer-to-peer systems due to its simpler file system semantics - only enough file system semantics are necessary to provide its upload and download capabilities. The prototype implementation does not support resumed upload. Fortress FTP builds a distributed file system using DKS that is accessible by all peers.

The file system does not store any file data. Only the directory structure and the location of the file blocks are stored in the distributed file system. Actual block storage is handled by the Block Storage component.

## 7.2 Authenticated Data Structure

Data authenticity in a peer-to-peer system should be strongly enforced in a peer-to-peer system because data is received from a less trusted source. We say less trusted source because in the context of this paper, the peers in the system are believed to be provided by a better trusted source than from an open system. This does not prevent from peers sending faulty data, however. Data authentication should be in place for any data intensive distributed systems.

Merkle trees [21] have been used in numerous systems to provide an authenticating data structure. Secure tree based data structures have been popular in distributed storage systems, most notably [10, 8, 18]. As such, we build a distributed file system structure for the FTP service to access.

Files are divided in 10 kB chunks and stored in the DHT. File system structure is developed using directory blocks, inode blocks, and data blocks much like the UNIX file system. Each block is hashed using the SHA1 hash. The hash value is the block's key. SHA1 hash is collision resistant so the blocks with even the smallest difference will have a different key value. Entries in directories block reference other blocks using these hash values, so the entire filesystem structure are said to be *self-verifying.*

Dividing files into blocks and distributing them over many peers has its advantages and disadvantages. On one hand, the file is well distributed among the peers achieving load balance, especially with large files. It also reduces the problem of a possible bottleneck point with a large and popular file. On the other hand, a lookup must be made for each data chunk which is a significant network traffic overhead whereas a file stored at a single location requires only one lookup.

One of the benefits of using secure hashes to blocks that blocks that have same contents will have the same hash value. This means that if a file has similar or same contents as other files in the system, the same blocks would not have to be uploaded again. When two blocks are identical, the content hashes will also be identical. Optimizing data transmission is critical in peer-to-peer systems where network connections are the most limited data path. The Data Block Storage component of Fortress FTP always checks if the data block already exists on the local disk before proceeding with the data transfer. Unnecessary transmission of data blocks can be avoided with the slight overhead of key bytes and communication overhead.

A drawback to using this structure for a file system is that an update to a deeply nested directory entry would require all of the directory blocks in the directory path to be updated. This means that for an update to be atomic, the all directory blocks making up the path of directory will require them to be written synchronously - any changes to the file-system structure would always require the root block to be updated. The cost of an update to a directory at $d$ depth would require $O(d \lg N)$. The cascading effect caused by an update is shown in Figure 7.2. Furthermore, multi-process writes to the directory structure will need a exclusive lock to the root directory block. Assuming the root block remains in place with a single key, then the root block could become a bottle neck point. In order to assure consistent state of the root block, the peer wishing to modify the root block would have to obtain a lock on it by using a distributed form of semaphore.

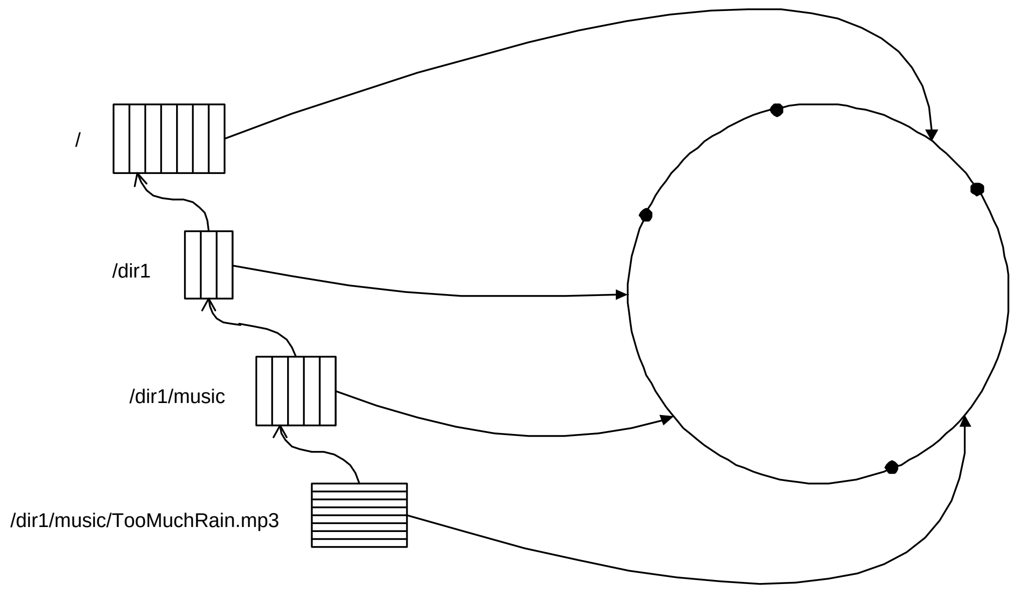

Figure 7.2: An update to a directory will cascade all the way up to the root block. All the blocks must be retrieved, updated, and reinserted into DKS. The directory block entries map to keys in DKS while the horizontal entries in the file block indicate entries that are retrieved through Data Block Storage module.

## 7.3 Data Availability

High data availability is a essential to an FTP service. High node availability increases data availability, but even with high availability of the nodes, the nodes cannot be expected to be available all the time. Data redundancy therefore is critical to the data availability in the design of the peer-to-peer FTP service. Data redundancy must be balanced according to the expected level of churn in the P2P system. We leave the degree of replication up to the users of the peer-to-peer group by leaving the redundancy rate as a configurable parameter.

Two popular methods for increasing data durability in peer-to-peer systems have been replication and erasure coding. Rodrigues and Liskov in [3] shows that because of the computational costs and complexity it introduces to the peer-to-peer system, erasure encoding should be avoided unless strictly required by the system design. Its main advantage is that it provides the same level of availability while using much less storage overhead (1 to 3-fold). Perhaps a more compelling reason for using erasure coding over replication is that the level of redundancy in replica-based system is much higher; maintenance traffic for the redundancy will be thus greater in replica-based systems.

## 7.4 Coping with Data Migration

When nodes join, leave, or fail in a distributed hash table, the key range responsibility must be adjusted to the correct nodes. In the case of DKS, newly joining nodes receive data for their key range from the predecessor. Conversely, when a node leaves the system, the node must move its data to its predecessor before leaving. Replicas are also migrated to proper peers when nodes join, leave, and fail. Essentially, dynamicity causes data migration in DKS.

In building applications over distributed hash tables, data migration caused by dynamicity should be carefully considered. Enough bandwidth should be available to handle the expected data migration due to dynamicity in a real deployment scenario. Another reason against storing so much data is that in F2F networks, node departures are not permanent and expected to be temporary. This means that if the replication feature of the DHT can mask the temporary departure of a few nodes, then the data migration for the replicas and the original copy can take place at a slow rate or at a later time. This can avoid large amounts of data transfers due to node departure/failure especially in a data intensive system where nodes are expected to hold a large amounts of data.

In Fortress FTP, the level of dynamicity in the system due to nodes failing and departing are expected to be low. Even so, each node is expected to hold a large quantity of data. This means that there would be a significant level of data migration and bandwidth consumption even when a single node leaves the system. Myriad Store [? ], a distributed backup application, reduces data migration due to dynamicity by storing only meta-data to data blocks instead of actually storing data blocks themselves in its distributed hash table.

Using a similar approach to Myriad Store, in Fortress FTP, only the directory structures are stored in the distributed hash table. Actual file transfers take place between peers using a transfer protocol between peers.

### 7.4.1 Data Block Replication

Data blocks must be replicated. If data blocks are not available during a download due to peer departure, whether it is permanent or temporary, downloads cannot complete. In Fortress FTP, a circular identifier space represents the keys to data blocks. Within this identifier space, every peer is assigned a contiguous range of data blocks they are responsible for.

Each data block is content hashed using SHA1 and the resulting 160-bit key is used as its identifier. Each peer also maintains $s$ successors' data blocks where $s$ is $lg\ N$ and $N$ is the number of peers in the system.

### 7.4.2 Data Maintenance

Data Maintenance is handled by the Data Block Storage layer. The data block storage layer employs lazy replication instead of eager replication as it is done in DKS to reduce data migration. Another reason for using lazy replication is that a peer departure may be temporary. This is particularly true in friendly networks where permanent departures are expected to be rare [13]. The Data Block Storage layer estimates the number of the nodes currently participating in the system, $N$, and determines the replication factor $r$. Currently, $r$ is determined to be $r = lg\ N$. The Data Block Storage layer attempts to maintain replicas at r successors.

## 7.5 Load balancing

FTP service is intrinsically data intensive. Early FTP server often became unavailable when client downloads exceeded the FTP upload capacity. Because Fortress FTP is peer-to-peer, failure of a single Fortress FTP server will not disrupt the collective peer-to-peer network. Dividing files into chunks also provides load balancing among the peers during file download. Because each Fortress FTP peer can provide an FTP service, the outgoing bandwidth of the FTP must also be balanced. We achieve this balance through the use of the PASV command. In active mode of FTP data transfer, the server connects to the client during data transfers. PASV (passive) mode, on the other hand, lets the client connect to the server for the data transfer - this is the preferred and default mode of most FTP clients in use today. In response to a PASV command, the server informs the client the server addresses and ports which the client can connect to in order to download the files.

In DrFTPD [9], the load is distributed by having a master server and several slave servers. Slave servers hold disjoint sets of files. When the RETR command is issued, the master server locates the slave server holding the file from its database. DrFTPD's the master server cannot anticipate which files the client will download, hence cannot determine which slave server address to inform the client when entering PASV mode. DrFTPD proposes and implements the PRET command as a solution to this problem. A few other disadvantages to this approach is that a single point of failure still exists at the master server and that slave servers can still become congested due to large and popular files.

In Fortress FTP all data is virtualized and available to every single Fortress FTP server. Having all the data available to the Fortress FTP server means that an Fortress FTP server can return any of the other Fortress FTP servers in response to the PASV command. In Fortress FTP, PASV command returns the address of the server that is able to provide the highest download rate to the client. Although in providing higher data availability and durability there is storage overhead, storage capacity is hardly an issue due to cheap and large storage devices.

## 7.6 Prototype Implementation

The authentication code for DKS and the Fortress FTP server is written entirely in Java. The implementation is based on the Java implementation of the DKS system. The code FTP server was based on the Apache FTP server. The interface to the native file system within the Apache FTP server was replaced with a distributed storage layer.

The key object in Fortress FTP for distributed storage is the DFSInputStream and the DFSOutputStream, which respectively extends the InputStream and the OutputStream interfaces. The encapsulation of these objects provide disk-like write and read operations through DKS and DataBlockStorage.

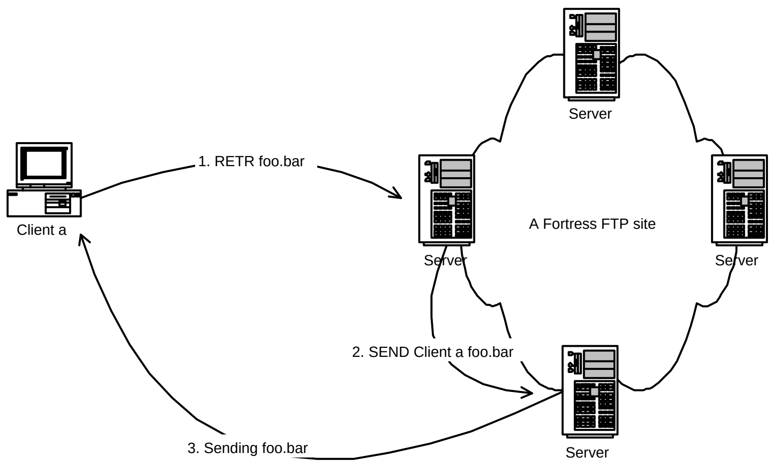

Figure 7.3: Client server interaction in active mode. In active mode the Fortress FTP servers can pick the server to send the file through election.

## 7.7 Using Fortress FTP

The command to begin the Fortress FTP server is:

```
% fftp keystore [-server on|off]
```

The Fortress FTP can be used in a number of ways. Even though every peer in the peer group can potentially become an FTP server, this is not strictly necessary. This is an administrative issue that is left up to the administrator(s). Peers can choose to deactivate the server functionality to mainly provide data storage and not the FTP service. This is particularly useful when a large amount of storage space is required on the server while the access to the FTP service is expected to be low.

The current implementation runs the FTP service as a separate and local service. Future improvements could involve the FTP servers to cooperate beyond the data distribution level. Having the FTP server as a separate program does have its advantages in that the users are managed locally and therefore giving each peer a local level of control to how the FTP service is accessed.

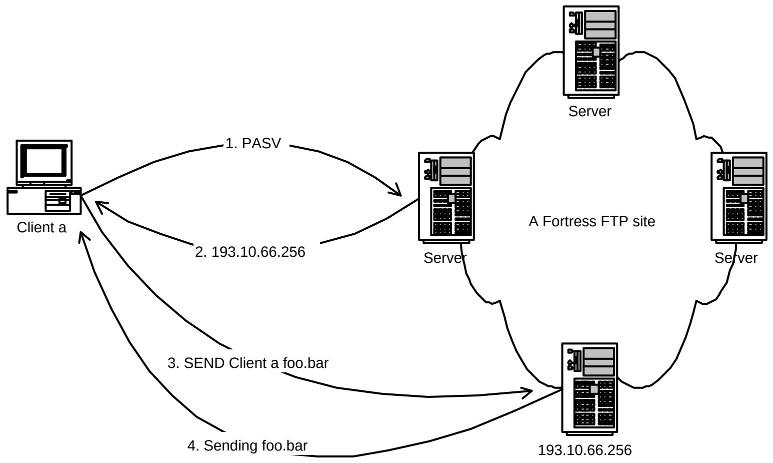

Figure 7.4: Client server interaction with PASV command. In passive mode the client is informed the address of the server that can provide the best service.

# Chapter 8

# Evaluation

Fortress FTP's transfer speeds are shown in figure /refevaltable1. The current implementation does not prefetch any of the blocks which explains the low transfer speeds. Without prefetching the overhead of connecting to a peer each time for a new block slows down the transfer process considerably. In order to further improve on the transfer rate, the socket connections could be kept alive for further block requests for a specific length of time. Caching of the blocks could improve the performance to a large extent for blocks that are frequently requested.

| | Upload speed | Download speed |
|---|---|---|
| Local Disk | 3.34 M bytes/s | 2.87 M bytes/s |
| Data Block Storage | 67.1 K bytes/s | 27.4 K bytes/s |

Table 8.1: Average transfer speeds for Data Block Storage and local disk.

As discussed in section /refxxftp, the depth of the directory tree has a direct impact on the length of time for the modification of a single directory block to be updated; modification to any directory block will require all of the blocks in the directory path to be updated. This effect is further exacerbated when directories contain large number of entries. See figure 8.2. This is a trade off in using a self authenticating structure.

Using cryptographic hash has its advantage in quasi-randomly distributing the hash value in the 160-bit identifier space. In a peer-to-peer system where tens of thousands or even millions of peers are expected to be present, this is a desirable solution. But in smaller instances of peer-to-peer systems where the numbers of peers do not exceed in several hundred, the distribution of the identifier space using cryptographic hash may not have enough even spread

| Depth | Time (ms) |
|---|---|
| 1 | 1365 |
| 5 | 3771 |
| 10 | 8541 |
| 50 | 42282 |
| 100 | 87149 |

Table 8.2: Time measurements for modifying directory at various depths.

in the identifier space. This is an issue that must be addressed in a real life deployment scenario where Fortress FTP sites may be composed of very few numbers of peers. Figure 8.3 depicts the distribution of identifier range with lower number of peers.

| Number of peers | Median of key range ratio | Std. Deviation |
|---|---|---|
| 5 | 0.243 | 0.100 |
| 50 | 0.015 | 0.017 |
| 150 | 0.004 | 0.007 |
| 500 | 0.001 | 0.002 |

Table 8.3: Sample distribution of key range for various peers in the system.

The graph in Figure 8.1 shows the distribution of a 1GB file over the peers in Fortress FTP. As expected, the variance of discrepancy between the amounts of data peers hold reduces with as the number of peer grows. Currently, the key range is assigned by cryptographically hashing the public key and the IP address of the peer. To reduce this discrepancy, another method of assigning key range to peers should be considered instead of static hashing.

Partitions in DKS have not been considered in the design of the Fortress FTP system. This is a significant issue in real life deployment scenario because partitions do occur and must be dealt with. In KESO distributed file system /citekeso, the modifications to file systems are merged. Changes made during a network partition are merged when the partitioned peer groups recombine. Similar approach can be applied to the Fortress FTP system.

Deletion of files from the Fortress FTP has not been addressed during the implementation. Since removal of the blocks from the peers is not critical in transferring of files, unused blocks can be garbage collected in a less eager fasion.

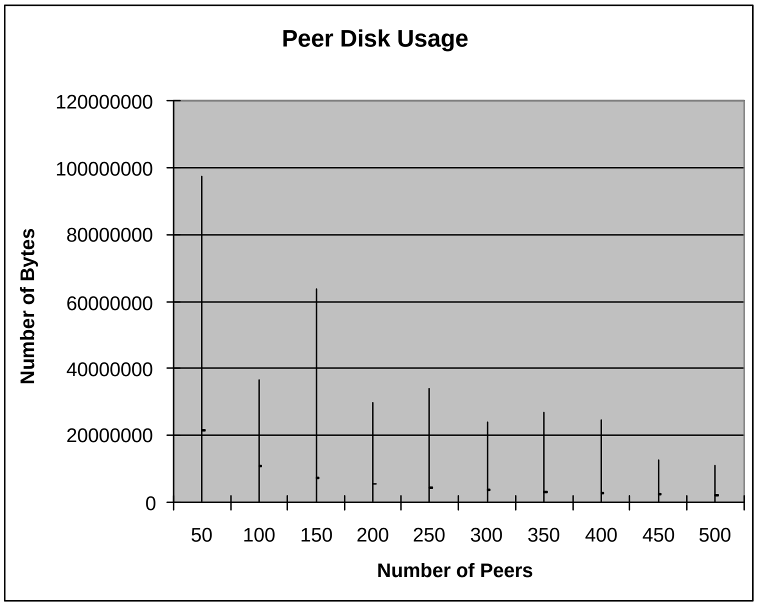

Figure 8.1: Peer disk usage for various number of peers.

# Chapter 9

# Conclusion

Features have been added to the DKS system that enable peers to form private peer-to-peer networks. Forming peer-to-peer networks this way allows peers in a group to select other peers that enter into the system. The expected result is that the membership to the private peer-to-peer networks will be governed by the peers already in the peer group. This concept is similar to the concept of F2F type of networks where connects are allowed only to trusted peers. The concept does differ from the F2F type of network in that it allows peers to invite their peers into the group - the newly invited peers may or may not be a friend to all peers already in the group. Although forming P2P networks this way is not purely F2F, it has its advantages in that it remains more scalable. Certainly if the group members are carefully managed by the members in the group, the private DKS can be used to form an instance of a F2F network. The loosened requirement of the private DKS can be utilized to form private overlay networks according to overlapping preferences of peer selection of individual peers. Users would then have to live with conflicting preferences or choose to resolve them through out-of-band methods.

Fortress FTP is just an example application of the private DKS. Any type of directory service or distributed applications that require key lookup is ideal for application of DHTs. Peer membership is verified the peer group by lookup up the peer's public key within the system. This lookup process uses the DKS to locate the peer's public key.

Currently, although the membership can be given or removed, no rules or algorithms regulate membership. That is, peers can always invite other peers to join the group, and any peer can be removed by any other peer. Incentive features or rules to membership could control the size of the peer group. For example, a peer's free rider behavior could further be discouraged if other peers were simply allowed to view other peers' performance. It would be interesting to see how cooperation in private networks can be *encouraged* rather than enforced through strict incentive mechanisms.

The group location service design sketches a system that help bootstrap peers to their private peer-to-peer network. The bootstrap information is encrypted to provide authenticity of the bootstrap information and to prevent non-members from accessing peer information.

# Bibliography

[1] E. Adar and B. Huberman. Free riding on gnutella, 2000.

[2] Seif Haridi Ali Ghodsi, Luc Onana Alima. Symmetirc replication for structured peer-to-peer systems, August 2005.

[3] Rodrigo Rodrigues And. High availability in dhts: Erasure coding vs. replication.

[4] W. Barry. An electronic group is virtually a social network, 1997.

[5] Bittorrent. http://www.bittorent.org.

[6] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–??, 2001.

[7] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[8] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[9] DrFTPD. http://drftpd.org.

[10] Peter Druschel and Antony Rowstron. PAST: A persistent and anonymous store. In *HotOS VIII*, May 2001.

[11] R. Dunbar. Coevolution of neocortical size, group size and language in humans. *Behavioral and Brain Sciences*, 16(4):681–735, 1993.

[12] M. Feldman, K. Lai, I. Stoica, and J. Chuang. Robust incentive techniques for peer-to-peer networks, 2004.

[13] Jinyang Li Frank. F2f: reliable storage in open networks.

[14] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Low-bandwidth topology maintenance for robustness in structured overlay networks. In *Proceedings of the 38st Annual Hawaii International Conference on System Sciences (HICSS).* IEEE Computer Society Press, 2004.

[15] Gnutella. http://rfc-gnutella.sourceforge.net.

[16] M. Gupta, P. Judge, and M. Ammar. A reputation system for peer-to-peer networks, 2003.

[17] Hales, D. and Patarin, S. How to cheat BitTorrent and why nobody does. *UBLCS*, 2005.

[18] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS.* ACM, November 2000.

[19] Per Brand Luc Onana Alima, Sameh El-Ansary and Seif Haridi. Dks(n, k, f): A family of low-communication, scalable and fault-tolerant infrastructures for p2p applications, 2003.

[20] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[21] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, 1979.

[22] Napster. http://www.napster.com.

[23] Overnet. http://www.edonkey2000.com/.

[24] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the Conference on File and Storage Technologies.* USENIX, 2003.

[25] WASTE. http://waste.sourceforge.net/.

[26] R. Zakon. FYI 32: Hobbes' Internet timeline, 2005.