DFI: Peer-to-Peer File Search

Will Huxtable
w@zif.io

March 26, 2017

Abstract

DFI is a peer-to-peer file sharing and indexing protocol. It is primarily for indexing and sharing BitTorrent infohashes and any associated metadata. Metadata includes things such as title, upload date, and file size. It uses an algorithm based on Kademlia to resolve peers, and actively tries to build its database of other nodes. It communicates using multiplexed TCP streams (similar to http/2), as these help provide NAT traversal. Authentication and signing uses Curve25519 keys, and node addresses are generated and encoded in a similar manner to Bitcoin addresses. DFI allows peers to both search each other remotely, or download a copy of the database for local searching. The protocol does not presently offer encryption, and those desiring this should use something such as Tor, I2P, cjdns, etc.

Contents

1 Introduction

While searching for webpages online has been easy since the advent of the search engine, searching specifically for files has not been quite as intuitive.

1.1 Current Solutions

When BitTorrent [1] was invented in 2001, suddenly we had a way of sharing files quickly, and without having to own or rent any sort of hosting infrastructure. The downloading of a torrent itself may be decentralised, yet to actually find the link to the file itself most users use a centralised index — something I have always found ironic.

The downsides of this are many:

censorship Any government, organization, or other controlling and potentially oppressive entity has the power to block access or totally remove an index. This can result in a vast loss of information.

traffic An index can see vast amounts of traffic flowing through it, making it expensive and difficult to run.

data loss For any of the above reasons, an index may go down. It could even just be an administration error. If it has no backups, access to files could potentially be lost forever.

discovery If you want people to be able to download your file, then you'd better put it on a popular index. This means that there are a few top indexes having a monopoly, leading to yet further centralisation.

1.2 DFI

DFI offers a new approach. Instead of having a database of magnet links and metadata in a datacentre somewhere, why not have it everywhere? Give everyone access to a means of sharing, without any limits.

Instead of one index indexing everyone's data, take a more peer-to-peer (p2p) approach. Make each node in the network store a database of its own files, have them index their own files. Take it a step further and allow any interested peers to download any other node's database, becoming both a seed for future downloads and a backup in case anything goes wrong. They also get the benefit of increased search speed. Peers can then answer both search queries for files, or download queries for their database. The former doesn't require a download step, but relies on the peer being available. The latter has greatly increased privacy, and doesn't rely on any other peers being online.

Peer details — such as name, address, public key, etc - are stored across the network. All that is needed is an address derived from the public key, and using Kademlia-style [2] lookups it can be resolved to the details needed.

Adding to the address resolution, DFI can actively explore the network to try and increase its knowledge of other peers.

To aid in address resolution, peers can store a database of peer details, reducing their own address resolution time and overall network bandwidth usage. We can then allow this database to be searched by users, allowing for easy content discovery.

1.3 Searching

DFI makes an important distinction between two different types of searching. If, for instance, we wish to search a peer's collection, it can happen in one of two ways.

remote make a connection, and send them a search query. This is more like a traditional index, and has minimal friction.

local this first requires downloading a peer's database, which may take a while depending on bandwidth. However, once this has been done, all searches can be performed very quickly, with no remote requests needed — which is also good for privacy.

1.3.1 Global search

Initially, the idea of everyone having full search access to all posts all the time seemed like a great idea. However, systems like Gnutella [3] have issues with malware and spam.

In a decentralised system, there is obviously no centralised control. One of the downsides of this is a complete lack of any sort of content moderation, weeding out the rubbish and ensuring that the content is real.

There is also the issue of copyrighted content spreading.

Overall, there would be the possibility that the network would become rife with fake, illegal, and generally dangerous downloads.

With the approach DFI has taken, we have to know the peer address in order to search them. While we could search the peer database for keywords, instead of fake content being there whether we want it to be or not, users are in charge of moderating their own content. If a node is filled with spam, users can just choose not to include it with their search results in the first place.

This approach has seen success in pretty much all subscription-based models.

1.4 Encryption and anonymity

DFI provides no encryption, no anonymity, and does not in fact route any connections or packets like other systems may. It relies on an underlying system for this sort of thing, and permits the use of SOCKS proxies to use such services. Below I will detail a few networks that could be used.

Tor [4] provides anonymity via Onion routing. It also encrypts by default, and aids in NAT traversal by using Onion addresses (.onion). However, it is slow, has high latency, and isn't entirely suited to p2p applications.

I2P [5] is similar to Tor, though with the intention of not accessing the "outer" normal internet. It also provides some level of anonymity and encryption. Its routing splits connections via many nodes, and every node in the network contributes. While it may be more suited to p2p, it has not been as studied or attacked as Tor has, and is slower due to a lack of popularity compared to Tor. It also provides NAT traversal.

cjdns [6] provides no anonymity, but it does provide an encrypted

IPv6 network, and is very fast and reliable. It should also be scalable, though it has not been studied as much as Tor. NAT traversal is also possible.

None of these methods of anonymity should be trusted for personal safety, especially against any sort of state-level actor.

It could be best to make cjdns a standard for DFI, as it provides all of the needed features, as well as causing no performance hits (it may potentially even increase performance and reliability).

The only downside is a lack of anonymity, though no users should **require** this and can gain some degree of privacy by not putting personal and identifying details in their `Entry`. Those needing it can use Tor.

2 Entries

DFI represents peers as Entries. These contain information about a peer, and can be used to connect to it, download a collection of the peer's posts, etc. Much like posts, I have found SQLite to be useful for storing entries.

Field	**Purpose**	**Limits**
Address	A DFI address, as specified below	
Name	The name of this node, specified by the user	64 chars
Desc	A brief description of this node, like a bio	256 chars
PublicAddress	The address we should connect to to access this node: could be IP, domain, onion, etc	256 chars
PublicKey	The Ed25519 public key for this `Entry`, used for verification	32 bytes
PostCount	The number of posts this `Entry` has	
Updated	When this node last updates it's `Entry`	
Signature	The signature for this `Entry`, used to verify all details are correct.	64 bytes
CollectionHash	The hash of a collection, defined in the next section	32 bytes
Port	The port this node listens for DFI collections on	max 65536
Seeds	a 2D array of bytes, each being a raw DFI address that seeds this node	
Seeding	a 2D array of bytes, each being a raw DFI address that this node is seeding	
Seen	When this node was last seen in the network	
Meta	A JSON-encoded object allowing for protocol extension	

2.1 Address

DFI has addresses that very much resemble Bitcoin addresses when encoded [7]. In fact, they are generated in pretty much the same way! Here is an example:

ZfkbZZXdHgLcmRoTqJjXpbZLtpWFLousS6

2.1.1 Features

Addresses are derived from an Ed25519 public key. This means that a node can ask any peer for the public key for a given address, and then derive an address from it. If this derivation is successful, then we know that we have recieved the correct public key. If not, then something is going wrong.

The binary version of an address is always 20 bytes, due to the hash function used. However, this is not particularly human-readable, so base58check encoding is used. This is slightly different to base64, and originally developed for use in Bitcoin. It has the benefit of avoiding ambiguous characters such as "0O" or "I1". Also seeing as it's entirely alphanumeric, various UX interactions work nicely with it.

2.1.2 Generating an address

Generating an address is actually very easy. DFI uses SHA3-256 and BLAKE2.

1. take the SHA3-256 sum of the public key
2. take the BLAKE2 sum of the previous sum
3. optionally base58check encode for human readability

So essentially, BLAKE2(SHA3-256(publicKey))

While similar to Bitcoin, DFI differs in choice of SHA hash function. SHA2 is a Merkle-Damgård hash, and therefore vulnerable to a length extension vulnerability. While a double hash solves this problem, it does introduce some other issues. SHA3, being in the Keccak family, does not suffer from a length extension flaw [8] — therefore making it the best choice for a new protocol.

When base58check encoding the address, it is prefixed with a "Z". This makes it obvious that this is a DFI address, and not anything else.

2.2 Signing

`Entries` must be signed in order to ensure that they can be shared and verified. Once the public key is known, it is easy to verify signatures. However, in order to do this we must first convert the `Entry` to a byte form in order to sign it. We can't just use JSON, as this does not guarantee any form of field ordering. We do not expect this format to ever be decoded, so no separators are needed. `Entries` are encoded by first converting each field to a string, then appending each string in order, like so:

1. name
2. description
3. public address
4. public key
5. port
6. post count
7. updated
8. collection hash
9. seeding [1]
10. meta

2.2.1 Mutability

You may have noticed that the `Entry` does not sign all of its fields. This is intentional! For instance, while it only makes sense for the `Entry` to change what it is seeding, this is not true for who seeds the `Entry`. New seeds can come about at any time, and it makes sense for all seeds to maintain a list of all other seeds, regardless of whether or not the node itself is online.

As well as that, signing the signature before replacing it would mean that entries would never validate.

Finally, the Seen field is not signed, as this should be updated by other nodes.

3 Posts

DFI is all about sharing posts. Posts represent an individual file or a collection of files, as well as any associated metadata. They can be stored in any way the client wishes, though I've found SQLite to be useful. It has a full text search feature which is particularly useful, see FTS4. In memory they can be easily represented as a struct.

[1] Each address we are seeding is stored as raw binary, not base58 encoded. They are also appended in the raw form.

Since a post contains a BitTorrent infohash, the files it is tied to can be downloaded using any BitTorrent client. Post searches can also be ordered by seeders and leechers, as these are indicative of popularity.

Field	Purpose
Id	starting at one and incrementing with every post: ids are local to a collection
InfoHash	a BitTorrent infohash, can be used to download the file
Title	the name of the file, used for searching
Size	the size of the file, in bytes
FileCount	how many files this post contains
Seeders	how many seeders the torrent file has
Leechers	how many leechers the torrent file has
UploadDate	the Unix timestamp (seconds) that this post was created at
Tags	a comma-separated list of tags relating to this post
Meta	a JSON object of metadata: useful for extensions

3.1 Pieces

DFI stores posts in groups called Pieces. Each `Piece` contains 1000 posts, and make it possible to verify posts. When downloading posts from a peer, it is done a piece at a time, as pieces do not need to be full — while they can contain 1000 posts, they could just contain one.

This carries on. Piece 0 may have 1000 posts, while piece 1 may only have 10.

A piece contains a checksum, and for this DFI uses SHA3-256. Every time a post is added to a piece, the checksum is updated. In this way, we can cryptographically verify that we have gotten all the posts we expect.

3.1.1 Post hashing

Because encoding methods such as JSON do not guarantee any sort of field ordering, which is required when generating a checksum, Posts must be encoded by DFI. When writing a post to a hashsum, the fields are written to a string in the order above. However, for reasons explained in 3.1.2, seeders and leechers are not included.

For instance:

```
Id|InfoHash|Title|Size|FileCount|UploadDate|Tags|Meta
```

3.1.2 Post immutability

Due to the fact that we may well be downloading a post database from a peer that did not author it, post data **must** be immutable and signed. Otherwise seeds would be free to edit titles and descriptions wherever they please, which could be confusing.

However, the only post field that should be able to change are the seeder and leecher count, as generally they tend to decrease over time. Because of this, they are not included in the hashing of a post for a piece checksum, meaning that any peer can change them.

When downloading a post from a seed, it would be prudent to check the seed/leech count with a BitTorrent tracker yourself, to avoid fakes.

3.2 Collections

Collections are, as the name suggests, collections of pieces. Essentially, each node in the network authors a single collection of pieces, and this collection may grow in time. Or, they may be empty.

Let's say Alice is the author of a collection, Eve is a seed for the collection, and Bob is attempting to download it from Eve. Eve could modify posts, and Bob would be none the wiser.

This is where collections come into play. Alice takes her pieces, and checksums each individual piece. Each checksum is then summed again, still using SHA3-256 - this forms a hashlist, with the final checksum being the root hash. Alice can then insert the root hash into her `Entry`, and sign the `Entry` with her private key.

Collections are made up of a root hash and a list of hashes. Since addresses can be used to verify the public key of an `Entry`, which can in turn be used to verify the `Entry` itself, Bob can fetch Alice's root hash from any node. These can then be used to fetch the hash list from anyone who is seeding Alice, in this case Eve. The hash list can be checksummed, and the resulting checksum compared to the root hash in Alice's `Entry`.

If the two checksums match, then Bob knows that he has a valid hash list.

Bob can then continue to download pieces from any of Alice's seeds, safe in the knowledge that they can all be verified via the hash list to be the same pieces that Alice authored.

With this in place, if Eve attempts to modify any piece or post, then checksums will no longer match, and Bob will know.

4 Protocol

DFI uses TCP for connections. This is to both make it easier to implement, and to make things less complicated. It also makes it easier to use SOCKS proxies, among other things.

To allow for multiple concurrent requests, as well as allowing for the recieving peer (or "server") to initiate connections, DFI uses stream multiplexing. Specifically the Yamux [9] library/protocol is used, which performs well.

The protocol allows for any listening port to be used, though 5050 is standard. This is mostly because it is often open in firewalls, and isn't used for anything important.

When initiating a DFI connection, a sequence of 4 bytes is expected. The first two are the DFI bytes, indicating that this is a DFI connection. The latter two are the protocol version. If the protocol versions differ, then the connection will be terminated. In this way, we can more easily distinguish DFI nodes of different versions in the future, without any weird protocol errors occuring. The DFI bytes are `0x7a66`, and the version is dependent on the client (though right now it is `0x0000`).

Once it is known that this is a DFI connection, handshaking can begin.

4.1 Messages

DFI uses the concept of messages to communicate. While it is not a datagram protocol, messages make it much easier to deal with requests and responses. A message is a struct containing a `Header` and a `Content` field. The former identifies the message type, which can be switched on for routing, while the latter allows for some form of data to be sent along with the request. The `Header` is a string, and the `Content` is anything that can be encoded to bytes.

Encoding wise, DFI uses MessagePack. This is more space-efficient than JSON, and is also more performant. The main downside is the loss in human readability for debugging, but tools can solve that. The message itself is MessagePack encoded, and any data intended for the `Content` field also is, by standard.

There are several message types defined, they are detailed in the table below. While it is possible for implementations to add their own message types, this should be discouraged as nodes will error on unknown messages. If extra functionality is required, it should be negotiated while handshaking.

Header	Purpose
header	indicates the start of a handshake: contains the `Entry` of the peer
cap	a struct detailing the "extra" capabilities of this peer, such as compression: what this contains is implementation-specific
ok	an affirmative response
no	a negative response, may also include an error message
term	indicates that the connection should be terminated, allowing for graceful shutdown
cookie	indicates that the message contains a cookie, used for proof of private key ownership
sig	indicates that the message contains a signature, used for sending back the signed cookie
done	whatever work that was happening has now completed
req.search	used to perform a remote search, contents should be the query
req.recent	requests the most recent posts a peer has
req.popular	requests the most popular posts a peer has
req.hashlist	requests a hashlist for a given peer, part of a collection
req.piece	requests a piece or range of pieces from a peer
req.addseed	requests that this node be added as a seed for a given peer
posts	the message contains posts, and is the usual response to something like req.recent
hashlist	the message contains a hashlist for a collection
dht.entry	the message contains a DHT `Entry`
dht.entries	the message contains a array of DHT entries
dht.query	query a peer for a `dht.entry`
dht.announce	inserts the nodes own `Entry` into another peer
dht.findclosest	request the closest entries to a given address

4.2 Error codes

DFI uses error codes when responding with a `err` message. The `Content` field of this message should contain an error code and optional detailing. The error codes specified here are taken from the HTTP error code specification.

Code	Purpose
300	used to indicate "multiple choices", or that a peer should refer to one of this nodes seeds.
400	bad request
403	forbidden
404	not found
500	internal server error
501	not implemented

4.3 Handshaking

From here and onwards, "server" shall refer to the peer recieving a connection, and "client" shall refer to the peer initiating a connection. The handshake process will be explained from the perspective of the server.

As soon as the server recieves a valid DFI TCP connection, it will begin handshaking. Once this occurs, the server expects a `header` message from the client, with its `Content` field containing the `Entry` of the client. If this is recieved without error, the client is sent `ok`, otherwise they are sent `no`.

After this, the server expects a `caps` message from the client, which details the capabilites of the client and can be used for negotiating things such as compression or encryption, or other protocol extensions. It has no predefined struct, and is essentially an arbitrary object — it is up to protocol implementations to define it.

Now that we have both the client capabilities and the client `Entry`, it is time for the client to prove that they are who they say they are. Essentially, they need to prove that they are in possession of the Ed25519 private key that corresponds to the public key in the `Entry`.

First, the server generates 20 random bytes. It is important that these are generated in a cryptographically secure manner. These are then sent to the client using a `cookie` message.

The client should then sign this cookie using their private key, and send the signature to the server using a `sig` message.

Now that it has the signature, the server verifies it against the public key and cookie that it already has. If there is a match, then the client is sent `ok`, otherwise it is sent `no`.

We are not quite done. Now it is time for roles to reverse: the client has proven who they are, but it is now the server's turn.

If the server is happy to continue, it will send the client `ok`, otherwise the client is sent `no`. If the client is sent `ok`, the server will also begin sending a handshake, going through the above process in the role of the client.

4.4 Message specification

4.4.1 req.search

This message asks a peer to perform a full text search upon all of the posts in its database. Note that a peer cannot (presently) handle remote search requests for any peers that it is a seed for, as search results are not cryptographically verified.

The `Content` field of this message contains a struct with fields of Query and `Page`. The `Query` is a string, essentially the text we want to search for. The `Page` is what page of results the client wishes to recieve.

Regardless of whether or not the peer has posts matching this query, it will still respond with `posts`. This is an array of post structs, and may well be empty.

4.4.2 req.recent and req.popular

These two messages are very similar, and very simple. The `Content` field contains nothing more than a page number.

Both respond with `posts`, and an array of posts. The only way they differ is that `req.recent` responds with the most recently uploaded posts and `req.popular` responds with the most popular posts.

What makes a post popular is entirely up to the client and the implementation, though using seeders and leechers is a good idea.

4.4.3 req.hashlist

This message sends a request for a hash list to a peer, for any peer. For the sakes of clarity, the peer we want the hash list for is called Bob, and the peer we are connected to is called Alice.

The `Content` field of a `req.hashlist` contains the binary of Bob's address. Alice will respond with the hash list if they have it, otherwise will respond with a `no` and an error.

Note that asking Alice for Bob's hash list should only really be done if we have any indication that Alice is a seed for Bob, otherwise there is no real reason for them to have the hash list.

4.4.4 req.piece

`req.piece` is used to request a piece to be downloaded from a peer. Like above, the peer who we want a piece for is called Bob, and the peer we are connected to is called Alice. It can also be used to request a range of pieces.

When Alice recieves a `req.piece`, it has the id of the start piece, how many pieces we should respond with, and the address of Bob.

These fields are called `Id`, `Length`, and `Address` respectively.

Alice will check if they have the database for Bob (may in fact be Alice). If not, Alice will respond with `no`. It is possible for us to query Alice with her own address, which should also work.

Alice will then load the posts for pieces starting with Id, until post with id `Id + Length`.

Once Alice has loaded the posts from their database, they will be sent to the peer. However, they are **not** MessagePack encoded, as this is wasteful. We already know the field names, and we already know the order they will be in. This is just a matter of copying the data. As such, posts are encoded in much the same way as they were when being hashed - however, seeders and leechers are in much the same manner, and inserted between `FileCount` and `UploadDate`.

Because at this point we are no longer sending messages, we need some other way of indicating that there are no longer any posts to send. To do this, send an encoded post with an id of -1. Because this is impossible otherwise, as ids must be positive, it can be used to indicate the end of a piece.

4.4.5 req.addseed

This message can be used to request that a peer add the sender as a seed for a given peer.

The `Content` field contains the raw DFI address of the peer that we want to become a seed for. The remote peer will then attempt to merge our address into the correct seed list, and will respond with `ok` if successful. Otherwise, the response will be `no` and an error will be the response.

4.4.6 dht.query

This is a pretty straightforward request. The client sends a message containing the raw DFI address it is looking for in the `Content` field, and the server attempts to look it up. If it has the `Entry`, it responds with `dht.entry` and an encoded `Entry`. Otherwise, the reponse is `no` and an error.

The server will also reinsert the result of the query into its routing table. This ensures that popular entries are more likely to show up in a `dht.findclosest`.

4.4.7 dht.findclosest

This request is mostly used for address resolution and exploration of the network.

Much like `dht.query` this message contains the raw DFI address in its `Content`.

The remote peer will then perform a lookup in the routing table, as described in the Address resolution section.

The results will be returned in a `dht.entries`. Otherwise, a `no` will be returned with an error.

4.4.8 dht.announce

This message takes an encoded `Entry` in the `Content` field. This is then inserted into the remote peer's NetDB, and the remote peer responds with either an `ok` on success or a `no` and error on failure.

4.5 NetDB

DFI has the concept of a "NetDB", which is essentially a distributed hash table, or DHT. It is more or less an implementation of Kademlia, though with some slightly different goals. Unlike Kademlia, each DFI node has the goal of **total** network knowledge — that is, all nodes know about all nodes.

Also unlike Kademlia, DFI has a separate backing database as well as a routing table. This does add additional complexity, but allows for greater functionality.

4.5.1 Separation of table and database

In order to allow for potential total network awareness, while still allowing for the routing table to remain entirely in memory, the database and routing table need to be separated. The NetDB is a combination of the two.

All known DFI nodes are stored in the main store, while a selection is stored in the routing table. It is recommended to take a random selection of the nodes in the store and add them to the table periodically, in order to keep data up-to-date.

To better understand some of the concepts that follow, I recommend you read the Kademlia paper [2]. Next I will be going over how NetDB works, as well as the Kademlia elements of it.

Like Kademlia, DFI uses the XOR distance metric. XOR is actually a proper distance metric, and satisfies the triangle inequality — it works great with binary addresses.

The routing table is represented as a selection of buckets, one bucket for each bit in the raw DFI address. The bucket size is known as k.

4.5.2 Address resolution

As has been explained prior, every DFI node has an assigned address, which is 20 bytes. If a node wishes to have the `Entry` for a peer, and it doesn't have that `Entry` in its own database, it will look to its routing table.

First, it takes the address as a byte array, and looks at each bit of each byte, starting with the leftmost. By doing this, we can count the number of "leading zeroes". This corresponds to a "bucket" in the routing table, containing all other DFI addresses with that number of leading zeroes. By the XOR metric, they are close to each other.

In order to find some nodes that may know about the address we are looking for, we look to the bucket that the address resides in. If the bucket is full, then great! Take the whole thing. If not, then start looking to the buckets either side, and repeat this until we have k closest nodes, or run out of buckets.

The resulting list of addresses can then be sorted by closeness to our target address.

We connect to the closest peer, and `dht.query` it for the target node. Hopefully it will respond with the `Entry` we are looking for. If not, then send it a `dht.findclosest`, which will cause it to perform the above lookup in its routing table and respond with the results. We can then recursively work through each `Entry` in this list.

The above process can be repeated with every other node in our initial list.

The main problem with this process is that if the node doesn't exist, or is barely known in the network, the number of queries and connections performed while trying to resolve it can be immense. Because of this it is probably best for implementations to limit the number of lookups.

4.6 Network exploration

This is something of a protocol extension. Clients do not need to implement this in order to function as a DFI node, however it is recommended to aid in network awareness.

Exploration should discover as many new nodes as possible via `dht.findclosest` requests.

In order to start, we need a node to work with. Ideally this would be chosen at random. The first step is to generate 20 random bytes, these can be treated as a DFI address. Send the node a `dht.findclosest` for the random bytes, and then follow up with another `dht.findclosest` for the client's own address. This means that a node both knows about those close to it which aids in address resolution, and as much of the network as possible.

Now that there is a set of results, the node can sort it. It's best to prefer nodes that have not been seen before, then nodes that have not been explored or seen in a while.

If an explore ever runs out of nodes to work with, then it can be seeded from the node's own routing table.

This explore job can be ran as often as every 2 minutes, though this should be configurable.

4.7 Periodic announcing

This is an opposite to an explore. While an explore seeks to fill the NetDB as much as possible, periodic announcing seeks to make sure that the rest of the network knows about our node. It is best if nodes announce to a `dht.findclosest` run on their own table for both their own address and a random one.

4.8 Seed discovery

Since nodes can have seeds, it is in their interest to try and gather as many of its seeds as possible into its seed list. After all, a node may become a seed for another without at all contacting the original author.

This should be done by contacting seeds in the seed list for an `Entry` periodically, and querying for the `Entry`. If the result has a different seed list, the two should be merged and any duplicates removed. This way, the seed list should be as full as possible.

Seeds should also go through this process, so that all seeds have access to the fullest list possible, and therefore know about each other.

4.9 Seed verification

It is possible to verify that an `Entry` is in fact a seed for another, instead of just blindly trusting the node we have recieved its `Entry` from.

The seed's `Entry` should be looked up, and its seeding list should be searched for the node it is allegedly seeding. If it is there, then we know that this is definitely a seed, as the seeding list is authenticated.

5 Future

This is not the final specification for DFI, and in the future I would like to add a few other features.

For one, I would like to allow for seeds to answer remote search queries on behalf of the original author. This would require some sort of seed "promotion" by the author, and could perhaps be done by some sort of friend system. A new list may have to be added to the `Entry`, detailing those the `Entry` trusts to answer queries on its behalf.

I would also like to allow for more file transfer protocols than just BitTorrent. This could be done by specifying some sort of Content-Type in posts, or a field in meta.

Another possible adjustment is culling of the NetDB. If DFI grows to be large, then there could be possible storage issues, where storing all known nodes becomes too many. A possible solution is removing any peer that has no content and has been inactive for a long time.

Finally, I would like to allow for user-submitted data — for instance, something like comments. This could be done in a similar manner to seed discovery. The only issues I can see here are those of spam, and any sort of user verification - UseNet has a similar issue with spam [10]. Sadly the best option here is some sort of hosted and centralised database for comments, as these are not essential to the network's functionality.

6 Conclusion

To conclude, I have presented a protocol for peer to peer search and indexing of BitTorrent infohashes. As of now, it is functional and I have a working implementation (https://github.com/dfi/dfi). DFI allows for nodes to leave and join at any time, and is resilient to censorship. Provided that a node has at least one connection to one node in the main network, it can still retain full network awareness. This full awareness is important as it allows address resolution to occur quickly, and minimises loss of information. I have not attempted to implement my own file transfer protocol: simply improve upon the indexing and discovery of files. The DFI protocol allows for this to be done in a decentralised manner, which allows for maximal sharing.

"Sharing is good, and with digital technology, sharing is easy."
Richard Stallman

References

[1] Bram Cohen and BitTorrent Inc (2001). *The BitTorrent Protocol Specification*. http://bittorrent.org/beps/bep_0003.html and https://en.wikipedia.org/wiki/BitTorrent for an overview.

[2] Petar Maymounkov and David Mazieres (2002). *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*. URL: http://www.maymounkov.org/papers/maymounkov-kademlia-lncs.pdf.

[3] Justin Frankel and Tom Pepper (2000). *Gnutella*. No official release, see wiki page. URL: https://en.wikipedia.org/wiki/Gnutella.

[4] Nick Mathewson Roger Dingledine and Paul Syverson (2004). *Tor: The Second-Generation Onion Router*. Usenix Security. URL: https://svn.torproject.org/svn/projects/design-paper/tor-design.pdf.

[5] I2P Team (2003). *I2P Technical Documentation*. URL: https://geti2p.net/en/docs.

[6] Caleb James DeLisle (cjdelisle) (2012). *cjdns Whitepaper*. URL: https://github.com/cjdelisle/cjdns/blob/master/doc/Whitepaper.md.

[7] Satoshi Nakamoto (2009). *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: https://bitcoin.org/bitcoin.pdf.

[8] Thai Duong and Juliano Rizzo (2009). *Flickr's API Signature Forgery Vulnerability*. URL: http://netifera.com/research/flickr_api_signature_forgery.pdf.

[9] *Yamux: Yet Another Multiplexer*. URL: https://github.com/hashicorp/yamux.

[10] *Newsgroup spam*. URL: https://en.wikipedia.org/wiki/Newsgroup_spam.