# Dat - Distributed Dataset Synchronization And Versioning

Maxwell Ogden, Karissa McKelvey, Mathias Buus

Version 1.0.0, May 2016

## Abstract

Dat is a swarm based version control system designed for sharing large datasets over networks such that their contents can be accessed randomly, be updated incrementally, and have the integrity of their contents be trusted. Every Dat user is simultaneously a server and a client exchanging pieces of data with other peers in a swarm on demand. As data is added to a Dat repository updated files are split into pieces based on Rabin fingerprinting and deduplicated against known pieces to avoid retransmission of data. File contents are automatically verified using secure hashes meaning you do not need to trust other nodes.

## 1. Introduction

There are countless ways to share share datasets over the Internet today. The simplest and most widely used approach, sharing files over HTTP, is subject to dead links when files are moved or deleted, as HTTP has no concept of history or versioning built in. E-mailing datasets as attachments is also widely used, and has the concept of history built in, but many email providers limit the maximum attachment size which makes it impractical for many datasets.

Cloud storage services like S3 ensure availability of data, but as they have a centralized hub-and-spoke networking model tend to be limited by their bandwidth, meaning popular files can be come very expensive to share. Services like Dropbox and Google Drive provide version control and synchronization on top of cloud storage services which fixes many issues with broken links but rely on proprietary code and infrastructure requiring users to store their data on cloud infrastructure which has implications on cost, transfer speeds, and user privacy.

Distributed file sharing tools like BitTorrent become faster as files become more popular, removing the bandwidth bottleneck and making file distribution effectively free. They also implement discovery systems which prevents broken links meaning if the original source goes offline other backup sources can be automatically discovered. However P2P file sharing tools today are not supported by Web browsers and do not provide a mechanism for updating files without redistributing a new dataset which could mean entire redownloading data you already have.

Decentralized version control tools for source code like Git provide a protocol for efficiently downloading changes to a set of files, but are optimized for text files and have issues with large files. Solutions like Git-LFS solve this by using HTTP to download large files, rather than the Git protocol. GitHub offers Git-LFS hosting but charges repository owners for bandwidth on popular files. Building a peer to peer distribution layer for files in a Git repository is difficult due to design of Git Packfiles which are delta compressed repository states that do not support random access to byte ranges in previous file versions.

Science is an example of an important community that would benefit from better approaches in this area. Increasingly scientific datasets are being provided online using one of the above approaches and cited in published literature. Broken links and systems that do not provide version checking or content addressability of data directly limit the reproducibility of scientific analyses based on shared datasets. Services that charge a premium for bandwidth cause monetary and data transfer strain on the users sharing the data, who are often on fast public university networks with effectively unlimited bandwidth. Version control tools designed for text files do not keep up with the demands of large data analysis in science today.

## 2. Inspiration

Dat is inspired by a number of features from existing systems.

### 2.1 Git

Git popularized the idea of a Merkle DAG, a way to represent changes to data where each change is addressed by the secure hash of the change plus all previous hashes. This provides a way to trust data integrity, as the only way a specific hash could be derived by another peer is if they have the same data and change history required to reproduce that hash. This is important for reproducibility as it lets you trust that a specific git commit hash refers to a specific source code state.

### 2.2 LBFS

LBFS is a networked file system that avoids transferring redundant data by deduplicating common regions of files and only transferring unique regions once. The deduplication algorithm they use is called Rabin fingerprinting and works by hashing the contents of the file using a sliding window and looking for content defined chunk boundaries that probabilistically appear at the desired byte offsets (e.g. every 1kb).

Content defined chunking has the benefit of being shift resistant, meaning if you insert a byte into the middle of a file only the first chunk boundary to the right of the insert will change, but all other boundaries will remain the same. With a fixed size chunking strategy, such as the one used by rsync, all chunk boundaries to the right of the insert will be shifted by one byte, meaning half of the chunks of the file would need to be retransmitted.

### 2.3 BitTorrent

BitTorrent implements a swarm based file sharing protocol for static datasets. Data is split into fixed sized chunks, hashed, and then that hash is used to discover peers that have the same data. An advantage of using BitTorrent for dataset transfers is that download bandwidth can be fully used. Since the file is split into pieces, and peers can efficiently discover which pieces each of the peers they are connected to have, it means one peer can download non-overlapping regions of the dataset from many peers at the same time in parallel, maximizing network throughput.

Fixed sized chunking has drawbacks for data that changes (see LBFS above). BitTorrent assumes all metadata will be transferred up front which makes it impractical for streaming or updating content. Most BitTorrent clients divide data into 1024 pieces meaning large datasets could have a very large chunk size which impacts random access performance (e.g. for streaming video).

### 2.4 Kademlia Distributed Hash Table

Kademlia is a distributed hash table, in other words a distributed key/value store that can serve a similar purpose to DNS servers but has no hard coded server addresses. All clients in Kademlia are also servers. As long as you know at least one address of another peer in the network, you can ask them for the key you are trying to find and they will either have it or give you some other people to talk to that are more likely to have it.

If you don't have an initial peer to talk to you have to use something like a bootstrap server that just randomly gives you a peer in the network to start with. If the bootstrap server goes down, the network still functions, and other methods can be used to bootstrap new peers (such as sending them peer addresses through side channels like how .torrent files include tracker addresses to try in case Kademlia finds no peers).

Kademlia is distinct from previous DHT designs such as Chord due to its simplicity. It uses a very simple XOR operation between two keys as its distance metric to decide which peers are closer to the data being searched for. On paper it seems like it wouldn't work as it doesn't take into account things like ping speed or bandwidth. Instead its design is very simple on purpose to minimize the amount of control/gossip messages and to minimize the amount of complexity required to implement it. In practice Kademlia has been extremely successful and is widely deployed as the "Mainline DHT" for BitTorrent, with support in all popular BitTorrent clients today.

### 2.5 Peer to Peer Streaming Peer Protocol (PPSPP)

PPSPP (IETF RFC 7574) is a protocol for live streaming content over a peer to peer network. In it they define a specific type of Merkle Tree that allows for subsets of the hashes to be requested by a peer in order to reduce the time-till-playback for end users. BitTorrent for example transfers all hashes up front, which is not suitable for live streaming.

Their Merkle trees are ordered using a scheme they call "bin numbering", which is a method for deterministically arranging an append-only log of leaf nodes into an in-order layout tree where non-leaf nodes are derived hashes. If you want to verify a specific node, you only need to request its sibling's hash and all its uncle hashes. PPSPP is very concerned with reducing round trip time and time-till-playback by allowing for many kinds of optimizations, such as to pack as many hashes into datagrams as possible when exchanging tree information with peers.

Although PPSPP was designed with streaming video in mind, the ability to request a subset of metadata from a large and/or streaming dataset is very desirable for many other types of datasets.

### 2.6 WebTorrent

With WebRTC browsers can now make peer to peer connections directly to other browsers. BitTorrent uses UDP sockets which aren't available to browser JavaScript, so can't be used as-is on the Web.

WebTorrent implements the BitTorrent protocol in JavaScript using WebRTC as the transport. This includes the BitTorrent block exchange protocol as well as the tracker protocol implemented in a way that can enable hybrid nodes, talking simultaneously to both BitTorrent and WebTorrent swarms (if a client is capable of making both UDP sockets as well as WebRTC sockets, such as Node.js). Trackers are exposed to web clients over HTTP or WebSockets.

### 2.7 InterPlanetary File System

IPFS also builds on many of the concepts from this section and presents a new platform similar in scope to the Web that has content integrity, peer to peer file sharing, version history and data permanence baked in as a sort of upgrade to the current Web. Whereas Dat is one application of these ideas that is specifically focused on sharing datasets but is agnostic to what platform it is built on, IPFS goes lower level and abstracts network protocols and naming systems so that any application built on the Web can alternatively be built on IPFS to inherit it's properties, as long as their hyperlinks can be expressed as content addressed addresses to the IPFS global Merkle DAG.

The research behind IPFS has coalesced many of these ideas into a more accessible format. We are still exploring how to best implement the Dat protocol on top of the IPFS platform.

## 3. DESIGN

Dat is a file sharing protocol that does not assume a dataset is static or that the entire dataset will be downloaded. The protocol is agnostic to the underlying transport e.g. you could implement Dat over carrier pigeon. The key properties of the Dat design are explained in this section.

- 1. **Mirroring** - All participants in the network simultaneously share and consume data.
- 2. **Content Integrity** - Data and publisher integrity is verified through use of signed hashes of the content.
- 3. **Parallel Transfer** - Subsets of the data can be accessed from multiple peers simultaneously, improving transfer speeds.
- 4. **Streaming Updates** - Datasets can be updated and distributed in real time to downstream peers.
- 5. **Secure Metadata** - Dat employs a capability system whereby anyone with a Dat link can connect to the swarm, but the link itself is a secure hash that is nearly impossible to guess and is never leaked by Dat itself.

## 3.1 Mirroring

Dat is a peer to peer protocol designed to exchange pieces of a dataset amongst a swarm of peers. As soon as a peer acquires their first piece of data in the dataset they become a partial mirror for the dataset. If someone else contacts them and needs the piece they have, they can share it. This can happen simultaneously while the peer is still downloading the pieces they want.

### 3.1.1 Source Discovery

An important aspect of mirroring is source discovery, the techniques that peers use to find each other. Source discovery means finding the IP and port of data sources online that have a copy of that data you are looking for. You can then connect to them and begin exchanging data using the Dat file exchange protocol, Hypercore. By using source discovery techniques we are able to create a network where data can be discovered even if the original data source disappears.

Source discovery can happen over many kinds of networks, as long as you can model the following actions:

- `join(key, [port])` - Begin performing regular lookups on an interval for `key`. Specify `port` if you want to announce that you share `key` as well.
- `leave(key, [port])` - Stop looking for `key`. Specify `port` to stop announcing that you share `key` as well.
- `foundpeer(key, ip, port)` - Called when a peer is found by a lookup

In the Dat implementation we implement the above actions on top of three types of discovery networks:

- DNS name servers - An Internet standard mechanism for resolving keys to addresses
- Multicast DNS - Useful for discovering peers on local networks
- Kademlia Mainline Distributed Hash Table - Zero point of failure, increases probability of Dat working even if DNS servers are unreachable

Additional discovery networks can be implemented as needed. We chose the above three as a starting point to have a complementary mix of strategies to increase the probability of source discovery.

Our implementation of peer discovery is called discovery-channel. We also run a custom DNS server that Dat clients use (in addition to specifying their own if they need to), as well as a DHT bootstrap server. These discovery servers are the only centralized infrastructure we need for Dat to work over the Internet, but they are redundant, interchangeable, never see the actual data being shared, anyone can run their own and Dat will still work even if they all are unavailable. If this happens discovery will just be manual (e.g. manually sharing IP/ports). Every data source that has a copy of the data also advertises themselves across these discovery networks.

### 3.1.2 Peer Connections

Up until this point we have just done searches to find who has the data we need. Now that we know who should talk to, we have to connect to them. Once we have a duplex binary connection to a peer we then layer on our own file sharing protocol on top, called Hypercore.

In our implementation, we use either TCP, UTP or WebRTC sockets for the actual peer to peer connections. UTP is nice because it is designed to *not* take up all available bandwidth on a network (e.g. so that other people sharing your wifi can still use the Internet). WebRTC support makes Dat work in modern web browsers using peer to peer connections.

When we get the IP and port for a potential source we try to connect using all available protocols and hope one works. If one connects first, we abort the other ones. If none connect, we try again until we decide that source is offline or unavailable to use and we stop trying to connect to them. Sources we are able to connect to go into a list of known good sources, so that if our Internet connection goes down we can use that list to reconnect to our good sources again quickly.

If we get a lot of potential sources we pick a handful at random to try and connect to and keep the rest around as additional sources to use later in case we decide we need more sources. A lot of these are parameters that we can tune for different scenarios later, but have started with some best guesses as defaults.

The connection logic is implemented in a module called discovery-swarm. This builds on discovery-channel and adds connection establishment, management and statistics. You can see stats like how many sources are currently connected, how many good and bad behaving sources you've talked to, and it automatically handles connecting and reconnecting to sources for you. Our UTP support is implemented in the module utp-native.

So now we have found data sources, connected to them, but we haven't yet figured out if they *actually* have the data we need. This is where our file transfer protocol Hyperdrive comes in. This is explained in a later section.

Peer connections types are outside the scope of the Dat protocol, but in the Dat implementation we make a best effort to make as many successful connections using our default types as possible. This means employing peer to peer connection techniques like UDP hole punching [?]. Our approach for UDP hole punching is to use a central known hole punching server which is accessible on the public Internet. In our implementation we re-use our custom DNS server by adding to it special functionality to facilitate peer message exchange for the purpose of hole punching.

In a scenario where two peers A and B want to connect, and both know the central server, this is how we perform UDP hole punching:

1. Peer A creates a local UDP socket and messages the central server that it is interested in connecting to people.
2. Central server messages Peer A back with a token that is a `hash(Peer A's remote IP + a local secret)`. The UDP packet contains the remote IP.
3. Peer A messages the central server with the token (this way you cannot spoof your IP and DDOS a remote peer)
4. Peer B does the same.
5. When the central server receives Peer B's message that it wants to connect to peers it forwards Peer B's message to Peer A and Peer A's message to Peer B.
6. Both peers now send a message to each other on their public IP and port. If UDP hole punching is supported by the routers of both peers at least one of the messages should get through.
7. At this point we reuse the UDP socket to run UTP on top to get a streaming reliable interface.

## 3.2 Content Integrity

Content integrity means being able to verify the data you received is the exact same version of the data that you expected. This is imporant in a distributed system as this mechanism will catch incorrect data sent by bad peers. It also has implications for reproducibility as it lets you refer to a specific version of the dataset you want.

A common issue in data analysis is when data changes but the link to the data remains the same. For example, one day a file called data.zip might change, but a simple HTTP link to the file does not include a hash of the content, so clients that only have the HTTP link have no way to check if the file changed. Looking up a file by the hash of its content is called content addressability, and lets users not only verify that the data they receive is the version of the data they want, but also lets people cite specific versions of the data by referring to a specific hash.

### 3.3 Parallel Transfer

### 3.4 Streaming Updates

### 3.5 Secure Metadata