Dat - Distributed Dataset Synchronization And Versioning

Maxwell Ogden, Karissa McKelvey, Mathias Buus

UNFINISHED, October 2016

Abstract

Dat is a protocol designed for sharing datasets over networks such that their contents can be accessed randomly or fully replicated, be updated incrementally and streamed, and have the integrity of their contents be trusted. Dat clients can simultaneously be uploading and/or downloading, exchanging pieces of data with other clients in a swarm on demand. Datasets can be multi-homed such that if the original source goes offline clients can choose to automatically discover additional sources. As data is added to a Dat repository, updated files are split into pieces using Rabin fingerprinting and deduplicated against known pieces to avoid retransmission of data. Data are automatically verified using secure hashes meaning data is protected against tampering or corruption. Dat guarantees privacy if the Dat Link is kept secret, but does not provide authentication of sources, only authentication of data.

1. Introduction

There are countless ways to 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 they have a centralized hub-and-spoke networking model and 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 services requiring users to store their data on cloud infrastructure which has implications on cost, transfer speeds, and user privacy.

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

Scientists are an example of a group that would benefit from better solutions to these problems. 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 that go unused. Version control tools designed for text files do not keep up with the demands of 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 directed acyclic graph (DAG) combined with a Merkle tree, a way to represent changes to data where each change is addressed by the secure hash of the change plus all ancestor hashes in a graph. 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.

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 distributed 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 easily support random access to byte ranges in previous file versions.

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 saturated. 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).

Another drawback of BitTorrent is due to the way clients advertise and discover other peers in absence of any protocol level privacy or trust. From a user privacy standpoint, BitTorrent leaks what users are accessing or attempting to access, and does not provide the same browsing privacy functions as systems like SSL.

2.4 Kademlia Distributed Hash Table

Kademlia is a distributed hash table, 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, most clients use a bootstrap server that randomly gives you a peer in the network to start with. If the bootstrap server goes down, the network still functions as long as 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 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.

Due to the simplicity in the original Kademlia design a number of attacks such as DDOS and/or sybil have been demonstrated. There are protocol extensions (BEPs) which in certain cases mitigate the effects of these attacks, such as BEP 44 which includes a DDOS mitigation technique. Nonetheless anyone using Kademlia should be aware of the limitations.

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 is a family of application and network protocols that have peer to peer file sharing and data permanence baked in. IPFS abstracts network protocols and naming systems to provide an alternative application delivery platform to todays Web. For example, instead of using HTTP and DNS directly, in IPFS you would use LibP2P streams and IPNS in order to gain access to the features of the IPFS platform.

2.8 Certificate Transparency/Secure Registers

The UK Government Digital Service have developed the concept of a register which they define as a digital public ledger you can trust. In the UK government registers are beginning to be piloted as a way to expose essential open data sets in a way where consumers can verify the data has not been tampered with, and allows the data publishers to update their data sets over time.

The design of registers was inspired by the infrastructure backing the Certificate Transparency project, initated at Google, which provides a service on top of SSL certificates that enables service providers to write certificates to a distributed public ledger. Anyone client or service provider can verify if a certificate they received is in the ledger, which protects against so called "rogue certificates".

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** - Any participant in the network can simultaneously share and consume data.
- 2. **Content Integrity** - Data and publisher integrity is verified through use of signed hashes of the content.
- 3. **Parallel Replication** - 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 other peers.
- 5. **End To End Encryption** - 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 difficult to guess.

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.

3.1.1.1 User Privacy

On the Web today, with SSL, there is a guarantee that the traffic between your computer and the server is private. As long as you trust the server to not leak your logs, attackers who intercept your network traffic will not be able to read the HTTP traffic exchanged between you and the server. This is a fairly straightforward model as clients only have to trust a single server for some domain.

There is an inherent tradeoff in peer to peer systems of source discovery vs. user privacy. The more people you ask for some data, the more people you trust to keep what your asked for private. Our goal is to have Dat be configurable in respect to this tradeoff to allow application developers to meet their own privacy guidelines.

It is up to client programs to make design decisions around which discovery networks they trust. For example if a Dat client decides to use the BitTorrent DHT to discover peers, and they are searching for a public Dat key with known contents, then because of the privacy design of the BitTorrent DHT anyone who can view that clients network traffic can find out what content they are searching for.

A client could choose to only use discovery networks with certain privacy guarantees. For example a client could only connect to an approved list of sources that they trust, similar to SSL. As long as they trust each source, the encryption built into the Dat network protocol will prevent the Dat key they are looking for from being leaked.

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, WebSockets or WebRTC for the network 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). WebSockets and WebRTC makes Dat work in modern web browsers.

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 Hypercore 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.

3.1.2.1 Hole Punching

When using raw UDP sockets 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 a dataset.

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 typical HTTP link to the file does not include a hash of the content, or provide a way to get updated metadata, so clients that only have the HTTP link have no way to check if the file changed without downloading the entire file again. Referring to 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.

Hypercore and Hyperdrive

Data storage and content integrity in Dat is implemented in a module called Hypercore. Given a stream of binary data, Hypercore splits the stream into chunks using Rabin fingerprints, hashes each chunk, and arranges the hashes in a specific type of Merkle tree that allows for certain replication properties. In addition to providing a content addressing system, Hypercore also provides a network protocol for exchanging chunks with peers. The defining feature of Hypercore is its ability to fully or partially synchronize streams in a distributed setting even if the stream is being appended to.

Hypercore is agnostic to the format of the input data, it operates on any stream of binary data. For the Dat use case of synchronizing datasets we wrote and use a file system module on top of Hypercore called Hyperdrive. We have a layered abstraction so that if someone wishes they can use Hypercore directly to have full control over how they model their data. Hyperdrive works well when your data can be represented as files on a filesystem, which is our main use case with Dat.

Registers

Central to the design of Hypercore is the notion of a register. This is a binary append-only stream (Kappa architecture) whose contents are cryptographically hashed and signed and therefore can be trusted. Hypercore lets you create many registers, and replicates them when synchronizing with another peer.

Registers are a way of encoding a Merkle tree that we use to efficiently replicate data over a network. When generating the Merkle tree, hashes are positioned by a scheme called binary interval numbering or just simply bin numbering. This is just a specific, deterministic way of laying out the nodes in a tree. For example a tree with 7 nodes will always be arranged like this:

```
0
  1
2
    3
4
  5
6
```

In our use case, the hashes of the actual content are always even numbers, at the wide end of the tree. So the above tree had four original values that become the even numbers:

```
value0 -> 0
value1 -> 2
value2 -> 4
value3 -> 6
```

All odd hashes are derived by hashing the two child nodes, e.g. given hash0 is `hash(value0)` and hash2 is `hash(value1)`, hash1 is `hash(hash0 + hash2)`.

A register contains two pieces of information:

1. List of binary values: [value0, value1, value2, ...]
2. List of hashes for the Merkle tree [hash0, hash1, hash2, ...]

The register itself interleaves these two lists such that the indexes (position) in the register are the same as the bin numbers from the Merkle tree. For example here is a register with three entries in pseudocode:

```
var feed = [{
  hash: sha256(value + size),
  size: value.length
  value: <some buffer>
}, {
  hash: sha256(feed[0].hash + feed[2].hash + size),
  size: feed[0].size + feed[1].size
}, {
  hash: sha256(value + size),
  size: value.length
  value: <some buffer>
}, {
  hash: sha256(feed[1].hash + feed[5].hash + size),
  size: feed[1].size + feed[5].size
}]
```

Registers can also be signed with a private key, allowing anyone with the corresponding public key to verify that new entries to the register were created by a holder of the private key. More on this in section 3.4.

3.3 Parallel Replication

Hypercore provides a replication protocol so two peers can communicate over a stateless messaging channel to discover and exchange data. Once you have received the register metadata, you can make individual requests for chunks from any peer you are connected to. This allows clients to parallelize data requests across the entire pool of peers they have established connections with.

Messages are encoded using Protocol Buffers. The protocol has nine message types:

Open

This should be the first message sent and is also the only message without a type. It looks like this:

```
message Open {
  required bytes feed = 1;
  required bytes nonce = 2;
}
```

The `feed` should be set to the discovery key as specified above. The `nonce` should be set to 24 bytes of high entropy random data. When running in encrypted mode this is the only message sent unencrypted.

`0` Handshake

This message is sent after sending an open message so it will be encrypted and we won't expose our peer id to a third party.

```
message Handshake {
  required bytes id = 1;
  repeated string extensions = 2;
}
```

`1` Have

Have messages give the other peer information about which blocks of data you have.

```
message Have {
  required uint64 start = 1;
  optional uint64 end = 2;
  optional bytes bitfield = 3;
}
```

You can use `start` and `end` to represent a range of data block bin numbers. If using a bitfield it should be encoded using a run length encoding described above. It is a good idea to send a have message soon as possible if you have blocks to share to reduce latency.

`2` Want

You can send a have message to give the other peer information about which blocks of data you want to have. It has type `2`.

```
message Want {
  required uint64 start = 1;
  optional uint64 end = 2;
}
```

You should only send the want message if you are interested in a section of the feed that the other peer has not told you about.

`3` Request

Send this message to request a block of data. You can request a block by block index or byte offset. If you are only interested in the hash of a block you can set the hash property to true. The nodes property can be set to a tree digest of the tree nodes you already have for this block or byte range. A request message has type `3`.

```
message Request {
  optional uint64 block = 1;
  optional uint64 bytes = 2;
  optional bool hash = 3;
  optional uint64 nodes = 4;
}
```

`4` Data

Send a block of data to the other peer. You can use this message to reply to a request or optimistically send other blocks of data to the other client. It has type `4`.

```
message Data {
  message Node {
    required uint64 index = 1;
    required uint64 size = 2;
    required bytes hash = 3;
  }

  required uint64 block = 1;
  optional bytes value = 2;
  repeated Node nodes = 3;
  optional bytes signature = 4;
}
```

5 Cancel

Cancel a previous sent request. It has type 5.

```
message Cancel {
  optional uint64 block = 1;
  optional uint64 bytes = 2;
}
```

6 Pause

An empty message that tells the other peer that they should stop requesting new blocks of data. It has type 6.

7 Resume

An empty message that tells the other peer that they can continue requesting new blocks of data. It has type 7.

3.4 Streaming Updates

3.5 Secure Metadata