Dat - Distributed Dataset Synchronization And Versioning

Maxwell Ogden, Karissa McKelvey, Mathias Buus

UNFINISHED, November 2016

Abstract

Dat is a protocol designed for syncing distributed, dynamic datasets. A secure changelog is used to ensure dataset versions are distributed safely. Files are efficiently versioned by checking new file regions against existing ones to duplication of existing similar file regions. Any byte range of any version of any file can be efficiently accessed as a stream from a Dat repository over a network connection. Consumers can choose to fully or partially replicate the contents of a remote Dat repository, and can also subscribe to live changes. Dat uses built-in public key cryptography to encrypt and sign all network traffic, allowing it to make certain privacy and security guarantees.

1. Background

Sharing datasets over the Internet is a subject of much study, but approaches remain relatively limiting. The 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 can prevent 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, do not have good privacy guarantees, and 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. Existing Work

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. Dat

Dat is a dataset synchronization 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. **Efficient Versioning** - Datasets can be efficiently synced, even in real time, to other peers using Dat Streams.
- 5. **Network Privacy** - Dat employs a capability system whereby anyone with a Dat link can connect to the swarm, but the link itself is very 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 can choose to become a partial mirror for the dataset. If someone else contacts them and needs the piece they have, they can choose to 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 a Dat Stream. By using source discovery techniques Dat is 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 four 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
- Signalhub - An HTTP key resolving service, non-distributed. Used by web browser clients who can't form raw UDP/TCP packets.

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

Our implementation of source 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).

TODO detail each discovery mechanism

3.1.2 Peer Connections

After the discovery phase, Dat should have a list of potential data sources to try and contact. Dat uses either TCP, UTP, WebSockets or WebRTC for the network connections. UTP is designed to not take up all available bandwidth on a network (e.g. so that other people sharing wifi can still use the Internet). WebSockets and WebRTC makes Dat work in modern web browsers. Note that these are the protocols we support in the reference Dat implementation, but the Dat protocol itself is transport agnostic.

When Dat gets the IP and port for a potential source it tries to connect using all available protocols and hopes one works. If one connects first, Dat aborts the other ones. If none connect, Dat will try again until it decides that source is offline or unavailable and then stops trying to connect to them. Sources Dat is able to connect to go into a list of known good sources, so that the Internet connection goes down Dat can use that list to reconnect to known good sources again quickly.

If Dat gets a lot of potential sources it picks a handful at random to try and connect to and keeps the rest around as additional sources to use later in case it decides it needs more sources.

The connection logic is implemented in a module called discovery-swarm. This builds on discovery-channel and adds connection establishment, management and statistics. It provides statistics such as how many sources are currently connected, how many good and bad behaving sources have been talked to, and it automatically handles connecting and reconnecting to sources. UTP support is implemented in the module utp-native.

Once a duplex binary connection to a remote source is open Dat then layers on its own protocol on top called a Dat Stream.

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.

Link rot, when links online stop resolving, and content drift, when when data changes but the link to the data remains the same, are two common issues in data analysis. 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.

Dat uses SHA256 hashes to address content. Hashes are arranged in a Merkle tree, a tree where each non-leaf node is the hash of all child nodes. Leaf nodes contain pieces of the dataset. This means that in order to verify the integrity of some subset of content only the top most common ancestors of the leaf nodes that contain that content must be fetched. For example to verify all content in a Merkle tree the top level node of the tree can be used. Due to the behavior of secure cryptographic hashes the top hash can only be produced if all data below it matches exactly. If two trees have matching top hashes then you know that all other nodes in the tree must match as well, and you can conclude that your dataset is synchronized.

3.2.1 Hypercore and Hyperdrive

The Dat storage, content integrity, and networking protocols are implemented in a module called Hy-

percore. 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 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.

3.2.2 Dat Streams

Dat Streams are binary append-only stream whose contents are cryptographically hashed and signed and therefore can be verified by anyone with access to the public key of the writer. They are an implemenation of the concept known as a register, a digital ledger you can trust. Dat lets you create many Streams, and replicates them when synchronizing with another peer.

Dat Streams use a specific method of encoding a Merkle tree where 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
```

A Dat Stream contains two pieces of information:

Evens: List of binary values with their hash and size: [value0, value1, value2, ...] Odds: List of Merkle hashes with the size of all their children: [hash0, hash1, hash2, ...]

These two lists get interleaved into a single register such that the indexes (position) in the register are the same as the bin numbers from the Merkle tree.

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)`.

For example a Dat Stream with two data entries would look something like this (pseudocode):

```
0. hash(value0)
1. hash(hash(value0) + hash(value1))
2. hash(value1)
```

3.3 Parallel Replication

Dat Streams include a message based replication protocol so two peers can communicate over a stateless channel to discover and exchange data. Once you have received the Stream 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 Efficient Versioning

Given a stream of binary data, Dat 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. Dat uses the chunk boundaries provided by Rabin fingerprinting to decide where to slice up the binary input stream. The Rabin implementation in Dat is tuned to produce a chunk every 16kb on average. This means for a 1MB file the initial chunking will produce around 64 chunks.

If a 1 byte edit is made to the file, chunking again should produce 63 existing chunks and 1 new chunk. This allows for deduplication of similar file regions across versions, which means Dat can avoid retransmitting or storing the same chunk twice even if it appears in multiple files.

Dat is also able to fully or partially synchronize streams in a distributed setting even if the stream is being appended to. This is accomplished by using the messaging protocol to traverse the Merkle tree of remote sources and fetch a strategic set of nodes. Due to the low level message oriented design of the replication protocol different node traversal strategies can be implemented.

TODO example of using protocol messages to request a subset of nodes in a live sync scenario

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

3.6 Network 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 sources you contact and ask for some data, the more sources you trust to keep what you 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 publicly shared Dat key with known contents, then because of the privacy design of the BitTorrent DHT it becomes public knowledge what key that client is 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.

3.6.2 Security

Dat links are Ed25519 public keys which have a length of 32 bytes (64 characters when Base64 encoded). Every Dat repository has corresponding a private key that kept internally in the Dat metadata and never shared.

Dat never exposes either the public or private key over the network. During the discovery phase the SHA256 hash of the public key is used as the discovery key. This means that the original key is impossible to discover (unless it was shared publicly through a separate channel) since only the hash of the key is exposed publicly.

All messages in the Dat protocol are encrypted using the public key during transport. This means that unless you know the public key (e.g. unless the Dat link was shared with you) then you will not be able to discover or communicate with any member of the swarm for that Dat. Anyone with the public key

can verify that messages (such as entries in a Dat Stream) were created by a holder of the private key.

Dat does not provide an authentication mechanism. Instead it provides a capability system. Anyone with the Dat link is currently considered able to discover and access data. Do not share your Dat links publicly if you do not want them to be accessed.