Dat - Distributed Dataset Synchronization And Versioning

Maxwell Ogden, Karissa McKelvey, Mathias Buus Madsen, Code for Science

May 2017

Abstract

Dat is a protocol designed for syncing folders of data, even if they are large or changing constantly. Dat uses a cryptographically secure register of changes to prove that the requested data version is distributed. A byte range of any file's version can be efficiently streamed 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. To ensure writer and reader privacy, Dat uses public key cryptography to encrypt network traffic. A group of Dat clients can connect to each other to form a public or private decentralized network to exchange data between each other. A reference implementation is provided in JavaScript.

1. Background

Many datasets are shared online today using HTTP and FTP, which lack built in support for version control or content addressing of data. This results in link rot and content drift as files are moved, updated or deleted, leading to an alarming rate of disappearing data references in areas such as published scientific literature.

Cloud storage services like S3 ensure availability of data, but they have a centralized hub-and-spoke networking model and are therefore limited by their bandwidth, meaning popular files can become 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 centralized cloud infrastructure which has implications on cost, transfer speeds, vendor lock-in 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 use link resolution and 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 entirely redownloading data you already have.

2. Dat

Dat is a dataset synchronization protocol that does not assume a dataset is static or that the entire dataset will be downloaded. The main reference implementation is available from npm as `npm install dat -g`.

The protocol is agnostic to the underlying transport e.g. you could implement Dat over carrier pigeon. Data is stored in a format called SLEEP (Ogden and Buus 2017), described in it's own paper. The key properties of the Dat design are explained in this section.

- 2.1 **Content Integrity** - Data and publisher integrity is verified through use of signed hashes of the content.
- 2.2 **Decentralized Mirroring** - Users sharing the same Dat automatically discover each other

and exchange data in a swarm.
- 2.3 **Network Privacy** - Dat provides certain privacy guarantees including end-to-end encryption.
- 2.4 **Incremental Versioning** - Datasets can be efficiently synced, even in real time, to other peers.
- 2.5 **Random Access** - Huge file hierarchies can be efficiently traversed remotely.

2.1 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 important 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 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 BLAKE2b (Aumasson et al. 2013) cryptographically secure hashes to address content. Hashes are arranged in a Merkle tree (Mykletun, Narasimha, and Tsudik 2003), a tree where each non-leaf node is the hash of all child nodes. Leaf nodes contain pieces of the dataset. Due to the properties 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. Trees are chosen as the primary data structure in Dat as they have a number of properties that allow for efficient access to subsets of the metadata, which allows Dat to work efficiently over a network connection.

Dat Links

Dat links are Ed25519 (Bernstein et al. 2012) public keys which have a length of 32 bytes (64 characters when Hex encoded). You can represent your Dat link in the following ways and Dat clients will be able to understand them:

- The standalone public key:

```
8e1c7189b1b2dbb5c4ec2693787884771201da9...
```

- Using the dat:// protocol:

```
dat://8e1c7189b1b2dbb5c4ec2693787884771...
```

- As part of an HTTP URL:

```
https://datproject.org/8e1c7189b1b2dbb5...
```

All messages in the Dat protocol are encrypted and signed 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.

Every Dat repository has a corresponding private key which is kept in your home folder and never shared. Dat never exposes either the public or private key over the network. During the discovery phase the BLAKE2b 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.

Dat does not provide an authentication mechanism at this time. 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.

Hypercore and Hyperdrive

The Dat storage, content integrity, and networking protocols are implemented in a module called Hypercore. 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.

Dat has a layered abstraction so that users 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 the main use case with Dat.

Hypercore Registers

Hypercore Registers are the core mechanism used in Dat. They are binary append-only streams 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 implementation of the concept known as a register, a digital ledger you can trust.

Dat uses two registers, `content` and `metadata`. The `content` register contains the files in your repository and `metadata` contains the metadata about the files including name, size, last modified time, etc. Dat replicates them both when synchronizing with another peer.

When files are added to Dat, each file gets split up into some number of chunks, and the chunks are then arranged into a Merkle tree, which is used later for version control and replication processes.

2.2 Decentralized 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 from others.

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. 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 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 - Less central points 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. Additionally you can specify a Dat via HTTPS link, which runs the Dat protocol in "single-source" mode, meaning the above discovery networks are not used, and instead only that one HTTPS server is used as the only peer.

Peer Connections

After the discovery phase, Dat should have a list of potential data sources to try and contact. Dat uses either TCP, HTTP or UTP (Rossi et al. 2010). UTP uses LEDBAT which 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), and is still based on UDP so works with NAT traversal techniques like UDP hole punching. HTTP is supported for compatibility with static file servers and web browser clients. Note that these are the protocols we support in the reference Dat implementation, but the Dat protocol itself is transport agnostic.

If an HTTP source is specified Dat will prefer that one over other sources. Otherwise when Dat gets the IP and port for a potential TCP or UTP source it tries to connect using both protocols. If one connects first, Dat aborts the other one. 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 if/when 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.

Once a duplex binary connection to a remote source is open Dat then layers on the Hypercore protocol, a message-based replication protocol that allows two peers to communicate over a stateless channel to request and exchange data. You open separate replication channels with many peers at once which allows clients to parallelize data requests across the entire pool of peers they have established connections with.

2.3 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 (e.g. a key cited publicly in a published scientific paper) 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.

2.4 Incremental Versioning

Given a stream of binary data, Dat splits the stream into chunks, hashes each chunk, and arranges the hashes in a specific type of Merkle tree that allows for certain replication properties.

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.

There are two types of versioning performed automatically by Dat. Metadata is stored in a folder called `.dat` in the root folder of a repository, and data is stored as normal files in the root folder.

Metadata Versioning

Dat tries as much as possible to act as a one-to-one mirror of the state of a folder and all its contents. When importing files, Dat uses a sorted, depth-first recursion to list all the files in the tree. For each file it finds, it grabs the filesystem metadata (filename, Stat object, etc) and checks if there is already an entry for this filename with this exact metadata already represented in the Dat repository metadata. If the file with this metadata matches exactly the newest version of the file metadata stored in Dat, then this file will be skipped (no change).

If the metadata differs from the current existing one (or there are no entries for this filename at all in the history), then this new metadata entry will be appended as the new 'latest' version for this file in the append-only SLEEP metadata content register (described below).

Content Versioning

In addition to storing a historical record of filesystem metadata, the content of the files themselves are also capable of being stored in a version controlled manner. The default storage system used in Dat stores the files as files. This has the advantage of being very straightforward for users to understand, but the downside of not storing old versions of content by default.

In contrast to other version control systems like Git, Dat by default only stores the current set of checked out files on disk in the repository folder, not old versions. It does store all previous metadata for old versions in `.dat`. Git for example stores all previous content versions and all previous metadata versions in the `.git` folder. Because Dat is designed for larger datasets, if it stored all previous file versions in `.dat`, then the `.dat` folder could easily fill up the users hard drive inadvertently. Therefore Dat has multiple storage modes based on usage.

Hypercore registers include an optional `data` file that stores all chunks of data. In Dat, only the `metadata.data` file is used, but the `content.data` file is not used. The default behavior is to store the current files only as normal files. If you want to run an 'archival' node that keeps all previous versions, you can configure Dat to use the `content.data` file instead. For example, on a shared server with lots of storage you probably want to store all versions. However on a workstation machine that is only accessing a subset of one version, the default mode of storing all metadata plus the current set of downloaded files is acceptable, because you know the server has the full history.

Merkle Trees

Registers in Dat use a specific method of encoding a Merkle tree where hashes are positioned by a scheme called binary in-order interval numbering or just "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 Dat, the hashes of the chunks of files are always even numbers, at the wide end of the tree. So the above tree had four original values that become the even numbers:

```
chunk0 -> 0
chunk1 -> 2
chunk2 -> 4
chunk3 -> 6
```

In the resulting Merkle tree, the even and odd nodes store different information:

- Evens - List of data hashes [chunk0, chunk1, chunk2, ...]
- Odds - List of Merkle hashes (hashes of child even nodes) [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(chunk0)` and hash2 is `hash(chunk1)`, hash1 is `hash(hash0 + hash2)`.

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

```
0. hash(value0)
1. hash(hash(chunk0) + hash(chunk1))
2. hash(value1)
```

It is possible for the in-order Merkle tree to have multiple roots at once. A root is defined as a parent node with a full set of child node slots filled below it.

For example, this tree hash 2 roots (1 and 4)

```
0
  1
2

4
```

This tree hash one root (3):

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

This one has one root (1):

```
0
  1
2
```

Replication Example

This section describes in high level the replication flow of a Dat. Note that the low level details are available by reading the SLEEP section below. For the sake of illustrating how this works in practice in a networked replication scenario, consider a folder with two files:

```
bat.jpg
cat.jpg
```

To send these files to another machine using Dat, you would first add them to a Dat repository by splitting them into chunks and constructing SLEEP files representing the chunks and filesystem metadata.

Let's assume `bat.jpg` and `cat.jpg` both produce three chunks, each around 64KB. Dat stores in a representation called SLEEP, but here we will show a pseudo-representation for the purposes of illustrating the replication process. The six chunks get sorted into a list like this:

```
bat-1
bat-2
bat-3
cat-1
cat-2
cat-3
```

These chunks then each get hashed, and the hashes get arranged into a Merkle tree (the content register):

```
0          - hash(bat-1)
    1      - hash(0 + 2)
2          - hash(bat-2)
        3  - hash(1 + 5)
4          - hash(bat-3)
    5      - hash(4 + 6)
6          - hash(cat-1)
8          - hash(cat-2)
    9      - hash(8 + 10)
10         - hash(cat-3)
```

Next we calculate the root hashes of our tree, in this case 3 and 9. We then hash them together, and cryptographically sign the hash. This signed hash now can be used to verify all nodes in the tree, and

the signature proves it was produced by us, the holder of the private key for this Dat.

This tree is for the hashes of the contents of the photos. There is also a second Merkle tree that Dat generates that represents the list of files and their metadata and looks something like this (the metadata register):

```
0 - hash({contentRegister: '9e29d624...'})
  1 - hash(0 + 2)
2 - hash({"bat.jpg", first: 0, length: 3})
4 - hash({"cat.jpg", first: 3, length: 3})
```

The first entry in this feed is a special metadata entry that tells Dat the address of the second feed (the content register). Note that node 3 is not included yet, because 3 is the hash of `1 + 5`, but 5 does not exist yet, so will be written at a later update.

Now we're ready to send our metadata to the other peer. The first message is a `Register` message with the key that was shared for this Dat. Let's call ourselves Alice and the other peer Bob. Alice sends Bob a `Want` message that declares they want all nodes in the file list (the metadata register). Bob replies with a single `Have` message that indicates he has 2 nodes of data. Alice sends three `Request` messages, one for each leaf node (`0, 2, 4`). Bob sends back three `Data` messages. The first `Data` message contains the content register key, the hash of the sibling, in this case node `2`, the hash of the uncle root `4`, as well as a signature for the root hashes (in this case `1, 4`). Alice verifies the integrity of this first `Data` message by hashing the metadata received for the content register metadata to produce the hash for node `0`. They then hash the hash `0` with the hash `2` that was included to reproduce hash `1`, and hashes their `1` with the value for `4` they received, which they can use the received signature to verify it was the same data. When the next `Data` message is received, a similar process is performed to verify the content.

Now Alice has the full list of files in the Dat, but decides they only want to download `cat.png`. Alice knows they want blocks 3 through 6 from the content register. First Alice sends another `Register` message with the content key to open a new replication channel over the connection. Then Alice sends three `Request` messages, one for each of blocks `4, 5, 6`. Bob sends back three `Data` messages with the data for each block, as well as the hashes needed to verify the content in a way similar to the process described above for the metadata feed.

2.5 Random Access

Dat pursues the following access capabilities:

- Support large file hierachies (millions of files in a single repository).
- Support efficient traversal of the hierarchy (listing files in arbitrary folders efficiently).
- Store all changes to all files (metadata and/or content).
- List all changes made to any single file.
- View the state of all files relative to any point in time.
- Subscribe live to all changes (any file).
- Subscribe live to changes to files under a specific path.
- Efficiently access any byte range of any version of any file.
- Allow all of the above to happen remotely, only syncing the minimum metadata necessary to perform any action.
- Allow efficient comparison of remote and local repository state to request missing pieces during synchronization.
- Allow entire remote archive to be synchronized, or just some subset of files and/or versions.

The way Dat accomplishes these is through a combination of storing all changes in Hypercore feeds, but also using strategic metadata indexing strategies that support certain queries efficiently to be performed by traversing the Hypercore feeds. The protocol itself is specified in Section 3 (SLEEP), but a scenario based summary follows here.

Scenario: Reading a file from a specific byte offset

Alice has a dataset in Dat, Bob wants to access a 100MB CSV called `cat_dna.csv` stored in the remote repository, but only wants to access the 10MB range of the CSV spanning from 30MB - 40MB.

Bob has never communicated with Alice before, and is starting fresh with no knowledge of this Dat repository other than that he knows he wants `cat_dna.csv` at a specific offset.

First, Bob asks Alice through the Dat protocol for the metadata he needs to resolve `cat_dna.csv` to the correct metadata feed entry that represents the file he wants. Note: In this scenario we assume Bob wants the latest version of `cat_dna.csv`. It is also possible to do this for a specific older version.

Bob first sends a `Request` message for the latest entry in the metadata feed. Alice responds. Bob looks at the `trie` value, and using the lookup algorithm described below sends another `Request` message for the metadata node that is closer to the filename he is looking for. This repeats until Alice sends Bob the matching metadata entry. This is the un-optimized resolution that uses `log(n)` round trips, though there are ways to optimize this by having Alice send additional sequence numbers to Bob that help him traverse in less round trips.

In the metadata record Bob recieved for `cat_dna.csv` there is the byte offset to the beginning of the file in the data feed. Bob adds his +30MB offset to this value and starts requesting pieces of data starting at that byte offset using the SLEEP protocol as described below.

This method tries to allow any byte range of any file to be accessed without the need to synchronize the full metadata for all files up front.

Scenario: Syncing live changes to files at a specific path

TODO

Scenario: Syncing an entire archive

TODO

3. Dat Network Protocol

The SLEEP format is designed to allow for sparse replication, meaning you can efficiently download only the metadata and data required to resolve a single byte region of a single file, which makes Dat suitable for a wide variety of streaming, real time and large dataset use cases.

To take advantage of this, Dat includes a network protocol. It is message-based and stateless, making it possible to implement on a variety of network transport protocols including UDP and TCP. Both metadata and content registers in SLEEP share the exact same replication protocol.

Individual messages are encoded using Protocol Buffers and there are ten message types using the following schema:

Wire Protocol

Over the wire messages are packed in the following lightweight container format

```
<varint - length of rest of message>
  <varint - header>
  <message>
```

The `header` value is a single varint that has two pieces of information: the integer `type` that declares a 4-bit message type (used below), and a channel identifier, `0` for metadata and `1` for content.

To generate this varint, you bitshift the 4-bit type integer onto the end of the channel identifier, e.g. `channel << 4 | <4-bit-type>`.

Feed

Type 0. Should be the first message sent on a channel.

- `discoveryKey` - A BLAKE2b keyed hash of the string 'hypercore' using the public key of the metadata register as the key.
- `nonce` - 32 bytes of random binary data, used in our encryption scheme

```
message Feed {
  required bytes discoveryKey = 1;
  optional bytes nonce = 2;
}
```

Handshake

Type 1. Overall connection handshake. Should be sent just after the feed message on the first channel only (metadata).

- `id` - 32 byte random data used as a identifier for this peer on the network, useful for checking if you are connected to yourself or another peer more than once
- `live` - Whether or not you want to operate in live (continuous) replication mode or end after the initial sync
- `userData` - User-specific metadata encoded as a byte sequence
- `extensions` - List of extensions that are supported on this Feed

```
message Handshake {
  optional bytes id = 1;
  optional bool live = 2;
  optional bytes userData = 3;
  repeated string extensions = 4;
}
```

Info

Type 2. Message indicating state changes. Used to indicate whether you are uploading and/or downloading.

Initial state for uploading/downloading is true. If both ends are not downloading and not live it is safe to consider the stream ended.

```
message Info {
  optional bool uploading = 1;
  optional bool downloading = 2;
}
```

Have

Type 3. How you tell the other peer what chunks of data you have or don't have. You should only send Have messages to peers who have expressed interest in this region with Want messages.

- `start` - If you only specify `start`, it means you are telling the other side you only have 1 chunk at the position at the value in `start`.
- `length` - If you specify length, you can describe a range of values that you have all of, starting from `start`.
- `bitfield` - If you would like to send a range of sparse data about haves/don't haves via bitfield, relative to `start`.

```
message Have {
  required uint64 start = 1;
  optional uint64 length = 2 [default = 1];
  optional bytes bitfield = 3;
}
```

When sending bitfields you must run length encode them. The encoded bitfield is a series of compressed and uncompressed bit sequences. All sequences start with a header that is a varint.

If the last bit is set in the varint (it is an odd number) then a header represents a compressed bit sequence.

```
compressed-sequence = varint(
  byte-length-of-sequence
  << 2 | bit << 1 | 1
)
```

If the last bit is *not* set then a header represents a non-compressed sequence.

```
uncompressed-sequence = varint(
  byte-length-of-bitfield << 1 | 0
) + (bitfield)
```

Unhave

Type 4. How you communicate that you deleted or removed a chunk you used to have.

```
message Unhave {
  required uint64 start = 1;
  optional uint64 length = 2 [default = 1];
}
```

Want

Type 5. How you ask the other peer to subscribe you to Have messages for a region of chunks. The `length` value defaults to Infinity or feed.length (if not live).

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

Unwant

Type 6. How you ask to unsubscribe from Have messages for a region of chunks from the other peer. You should only Unwant previously Wanted regions, but if you do Unwant something that hasn't been Wanted it won't have any effect. The `length` value defaults to Infinity or feed.length (if not live).

```
message Unwant {
  required uint64 start = 1;
  optional uint64 length = 2;
}
```

Request

Type 7. Request a single chunk of data.

- `index` - The chunk index for the chunk you want. You should only ask for indexes that you have received the Have messages for.
- `bytes` - You can also optimistically specify a byte offset, and in the case the remote is able to resolve the chunk for this byte offset depending on their Merkle tree state, they will ignore the `index` and send the chunk that resolves for this byte offset instead. But if they cannot resolve the byte request, `index` will be used.
- `hash` - If you only want the hash of the chunk and not the chunk data itself.
- `nodes` - A 64 bit long bitfield representing which parent nodes you have.

The `nodes` bitfield is an optional optimization to reduce the amount of duplicate nodes exchanged during the replication lifecycle. It indicates which parents you have or don't have. You have a maximum of 64 parents you can specify. Because `uint64` in Protocol Buffers is implemented as a varint, over the wire this does not take up 64 bits in most cases. The first bit is reserved to signify whether or not you need a signature in response. The rest of the bits represent whether or not you have (`1`) or don't have (`0`) the information at this node already. The ordering is determined by walking parent, sibling up the tree all the way to the root.

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

Cancel

Type 8. Cancel a previous Request message that you haven't received yet.

```
message Cancel {
  required uint64 index = 1;
  optional uint64 bytes = 2;
  optional bool hash = 3;
}
```

Data

Type 9. Sends a single chunk of data to the other peer. You can send it in response to a Request or unsolicited on its own as a friendly gift. The data

includes all of the Merkle tree parent nodes needed to verify the hash chain all the way up to the Merkle roots for this chunk. Because you can produce the direct parents by hashing the chunk, only the roots and 'uncle' hashes are included (the siblings to all of the parent nodes).

- `index` - The chunk position for this chunk.
- `value` - The chunk binary data. Empty if you are sending only the hash.
- `Node.index` - The index for this chunk in in-order notation
- `Node.hash` - The hash of this chunk
- `Node.size`- The aggregate chunk size for all children below this node (The sum of all chunk sizes of all children)
- `signature` - If you are sending a root node, all root nodes must have the signature included.

```
message Data {
  required uint64 index = 1;
  optional bytes value = 2;
  repeated Node nodes = 3;
  optional bytes signature = 4;

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

4. Multi-Writer

The design of Dat up to this point assumes you have a single keyholder writing and signing data and appending it to the metadata and content feed. However having the ability for multiple keyholders to be able to write to a single repository allows for many interesting use cases such as forking and collaborative workflows.

In order to do this, we use one `metadata.data` feed for each writer. Each writer kets their own keypair. Each writer is responsible for storing their private key. To add a new writer to your feed, you include their key in a metadata feed entry.

For example, if Alice wants to add Bob to have write access to a Dat repository, Alice would take Bob's public key and writes it to the 'local' metadata feed (the feed that Alice owns, e.g. the original feed). Now anyone else who replicates from Alice will find Bob's key in the history. If in the future Bob distributes a version of the Dat that he added new data to, everyone who has a copy of the Dat from Alice will have a copy of Bob's key that they can use to verify that Bob's writes are valid.

On disk, each users feed is stored in a separate hyperdrive. The original hyperdrive (owned by Alice) is called the 'local' hyperdrive. Bob's hyperdrive would be stored separately in the SLEEP folder addressed by Bob's public key.

In case Bob and Alice write different values for the same file (e.g. Bob creates a "fork"), when they sync up with each other replication will still work, but for the forked value the Dat client will return an array of values for that key instead of just one value. The values are linked to the writer that wrote them, so in the case of receiving multiple values, clients can choose to choose the value from Alice, or Bob, or the latest value, or whatever other strategy they prefer.

If a writer updates the value of a forked key with new value they are performing a merge.

5. Existing Work

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

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.

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. 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.

Kademlia Distributed Hash Table

Kademlia (Maymounkov and Mazieres 2002) 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.

Peer to Peer Streaming Peer Protocol (PPSPP)

PPSPP (IETF RFC 7574, (Bakker, Petrocco, and Grishchenko 2015)) 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.

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.

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 today's 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.

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 (Laurie, Langley, and Kasper 2013) project, initiated at Google, which provides a service on top of SSL certificates that enables service providers to write certificates to a distributed public ledger. Any client or service provider can verify if a certificate they received is in the ledger, which protects against so called "rogue certificates".

6. Reference Implementation

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.

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).

Acknowledgements

This work was made possible through grants from the John S. and James L. Knight and Alfred P. Sloan Foundations.

References

Aumasson, Jean-Philippe, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. 2013. "BLAKE2: Simpler, Smaller, Fast as Md5." In *International Conference on Applied Cryptography and Network Security*, 119–35. Springer.

Bakker, A, R Petrocco, and V Grishchenko. 2015. "Peer-to-Peer Streaming Peer Protocol (Ppspp)."

Bernstein, Daniel J, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. "High-Speed High-Security Signatures." *Journal of Cryptographic Engineering*. Springer, 1–13.

Laurie, Ben, Adam Langley, and Emilia Kasper. 2013. "Certificate Transparency."

Maymounkov, Petar, and David Mazieres. 2002. "Kademlia: A Peer-to-Peer Information System Based on the Xor Metric." In *International Workshop on Peer-to-Peer Systems*, 53–65. Springer.

Mykletun, Einar, Maithili Narasimha, and Gene Tsudik. 2003. "Providing Authentication and Integrity in Outsourced Databases Using Merkle Hash Trees." *UCI-SCONCE Technical Report*.

Ogden, Maxwell, and Mathias Buus. 2017. "SLEEP - the Dat Protocol on Disk Format." In.

Rossi, Dario, Claudio Testa, Silvio Valenti, and Luca Muscariello. 2010. "LEDBAT: The New Bittorrent Congestion Control Protocol." In *ICCCN*, 1–6.