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. A cryptographically secure register of changes to the data over time is used to ensure dataset versions are distributed as exact copies of exact versions. Any byte range of any version of any file 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. Dat uses built-in public key cryptography to encrypt network traffic, allowing it to make certain privacy and security guarantees. A group of Dat clients can connect to each other to form a public or private decentralized network to exchange data amognst each other in a swarm. 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 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 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 entire 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 from npm as `npm install dat -g`.

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.

- 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 exhange 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.

2.1 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 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 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 corresponding a private key that 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 implemenation 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 processed.

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 support 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 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 it's 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 inadverntently. 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 `cat.jpg` produces three chunks, and `bat.jpg` produces four 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 seven 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()
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.png", first: 0, last: 3})
4 - hash({"cat.png", first: 3, last: 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 signature they received 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.

3. SLEEP Specification

This section is a technical description of the SLEEP format intended for implementers. SLEEP is the the on-disk format that Dat produces and uses. It is a set of 9 files that hold all of the metadata needed to list the contents of a Dat repository and verify the integrity of the data you receive. SLEEP is designed to work with REST, allowing servers to be plain HTTP file servers serving the static SLEEP files, meaning you can implement a Dat protocol client using HTTP with a static HTTP file server as the backend.

SLEEP files contain metadata about the data inside a Dat repository, including cryptographic hashes, cryptographic signatures, filenames and file permissions. The SLEEP format is specifically engineered to allow efficient access to subsets of the metadat and/or data in the repository, even on very large repositories, which enables Dat's peer to peer networking to be fast.

The acronym SLEEP is a slumber related pun on REST and stands for Syncable Lightweight Event Emitting Persistence. The Event Emitting part refers to how SLEEP files are append-only in nature, meaning they grow over time and new updates can be subscribed to as a realtime feed of events through the Dat protocol.

The SLEEP version described here, used in Dat as of 2017 is SLEEP V2. SLEEP V1 is documented at http://specs.okfnlabs.org/sleep.

SLEEP Files

SLEEP is a set of 9 files that should be stored with the following names. In Dat, the files are stored in a folder called `.dat` in the top level of the repository.

```
metadata.key
metadata.signatures
metadata.bitfield
metadata.tree
metadata.data
content.key
content.signatures
content.bitfield
content.tree
```

The files prefixed with `content` store metadata about the primary data in a Dat repository, for example the raw binary contents of the files. The files prefixed with `metadata` store metadata about the files in the repository, for example the filenames, file sizes, and file permissions. The `content` and `metadata` files are both Hypercore registers, making SLEEP a set of two Hypercore registers.

SLEEP File Headers

The following structured binary format is used for `signatures`, `bitfield`, and `tree` files. The header contains metadata as well as information needed to decode the rest of the files after the header. SLEEP files are designed to be easy to append new data to at the end, easy to read arbitrary byte offsets in the middle, and are relatively flat, simple files that rely on the filesystem for the heavy lifting.

SLEEP files are laid out like this:

```
<32 byte header>
<fixed-size entry 1>
<fixed-size entry 2>
<fixed-size entry ...>
<fixed-size entry n>
```

- 32 byte header
- 4 bytes - magic byte (value varies depending on which file, used to quickly identify which file type it is)
- 1 byte - version number of the file header protocol, current version is 0
- 2 byte Uint16BE - entry size, describes how long each entry in the file is
- 1 byte - length prefix for body
- rest of 32 byte header - string describing key algorithm (in dat 'ed25519'). length of this string matches the length in the previous length prefix field. This string must fit within the 32 byte header limitation (24 bytes reserved for string). Unused bytes should be filled with zeroes.

Possible values in the Dat implementation for the body field are:

```
Ed25519
BLAKE2b
```

To calculate the offset of some entry position, first read the header and get the entry size, then do `32 + entrySize * entryIndex`. To calculate how many entries are in a file, you can use the entry size and the filesize on disk and do `(fileSize - 32) / entrySize`.

As mentioned above, `signatures`, `bitfield` and `tree` are the three SLEEP files. There are two additional files, `key`, and `data`, which do not contain SLEEP file headers and store plain serialized data for easy access. `key` stores the public key that is described by the `signatures` file, and `data` stores the raw chunk data that the `tree` file contains the hashes and metadata for.

File Descriptions

key

The public key used to verify the signatures in the `signatures` file. Stored in binary as a single buffer written to disk. To find out what format of key is stored in this file, read the header of `signatures`. In Dat, it's always a ed25519 public key, but other implementations can specify other key types using a string value in that header.

tree

A SLEEP formatted 32 byte header with data entries representing a serialized merkle tree based on the data in the data storage layer. All the fixed size nodes written in in-order tree notation. The header

algorithm string for `tree` files is `BLAKE2b`. The entry size is 40 bytes. Entries are formatted like this:

```
<32 byte header>
  <4 byte magic string: 0x05025702>
  <1 byte version number: 0>
  <2 byte entry size: 40>
  <1 byte algorithm name length prefix: 7>
  <7 byte algorithm name: BLAKE2b>
  <17 zeroes>
<40 byte entries>
  <32 byte BLAKE2b hash>
  <8 byte Uint64BE children leaf byte length>
```

The children leaf byte length is the byte size containing the sum byte length of all leaf nodes in the tree below this node.

This file uses the in-order notation, meaning even entries are leaf nodes and odd entries are parent nodes (non-leaf).

To prevent pre-image attacks, all hashes start with a one byte type descriptor:

```
0 - LEAF
1 - PARENT
2 - ROOT
```

To calculate leaf node entries (the hashes of the data entries) we hash this data:

```
BLAKE2b(
  <1 byte type>
    0
  <8 bytes Uint64BE>
    length of entry data
  <entry data>
)
```

Then we take this 32 byte hash and write it to the tree as 40 bytes like this:

```
<32 bytes>
  BLAKE2b hash
<8 bytes Uint64BE>
  length of data
```

Note that the Uint64 of length of data is included both in the hashed data and written at the end of the entry. This is to expose more metadata to Dat for advanced use cases such as verifying data length in sparse replication scenarios.

To calculate parent node entries (the hashes of the leaf nodes) we hash this data:

```
BLAKE2b(
  <1 byte>
    1
  <8 bytes Uint64BE>
    left child length + right child length
  <32 bytes>
    left child hash
  <32 bytes>
    right child hash
)
```

Then we take this 32 byte hash and write it to the tree as 40 bytes like this:

```
<32 bytes>
  BLAKE2b hash
<8 bytes Uint64BE>
  left child length + right child length
```

The reason the tree entries contain data lengths is to allow for sparse mode replication. Encoding lengths (and including lengths in all hashes) means you can verify the merkle subtrees independent of the rest of the tree, which happens during sparse replication scenarios.

The tree file corresponds directly to the `data` file.

data

The `data` file is only included in the SLEEP format for the `metadata.*` prefixed files which contains filesystem metadata and not actual file data. For the `content.*` files, the data is stored externally (in Dat it is stored as normal files on the filesystem and not in a SLEEP file). However you can configure Dat to use a `content.data` file if you want and it will still work. If you want to store the full history of all versions of all files, using the `content.data` file would provide that guarantee, but would have the disadvantage of storing files as chunks merged into one huge file (not as user friendly).

The `data` file does not contain a SLEEP file header. It just contains a bunch of concatenated data entries. Entries are written in the same order as they appear in the `tree` file. To read a `data` file, first decode the `tree` file and for every leaf in the `tree` file you can calculate a data offset for the data described by that leaf node in the `data` file.

Index Lookup

For example, if we wanted to seek to a specific entry offset (say entry 42):

- First, read the header of the `tree` file and get the entry size, then do `32 + entrySize * 42` to get the raw tree index: `32 + (40 * 42)`
- Since we want the leaf entry (even node in the in-order layout), we multiply the entry index by 2: `32 + (40 * (42 * 2))`
- Read the 40 bytes at that offset in the `tree` file to get the leaf node entry.
- Read the last 8 bytes of the entry to get the length of the data entry
- To calculate the offset of where in the `data` file your entry begins, you need to sum all the lengths of all the earlier entries in the tree. The most efficient way to do this is to sum all the previous parent node (non-leaf) entry lengths. You can also sum all leaf node lengths, but parent nodes contain the sum of their childrens lengths so it's more efficient to use parents. During Dat replication, these nodes are fetched as part of the Merkle tree verification so you will already have them locally. This is a log(N) operation where N is the entry index. Entries are also small and therefore easily cacheable.
- Once you get the offset, you use the length you decoded above and read N bytes (where N is the decoded length) at the offset in the `data` file. You can verify the data integrity using the 32 byte hash from the `tree` entry.

Byte Lookup

The above method illustrates how to resolve a chunk position index to a byte offset. You can also do the reverse operation, resolving a byte offset to a chunk position index. This is used to stream arbitrary random access regions of files in sparse replication scenarios.

- First, you start by calculating the current Merkle roots
- Each node in the tree (including these root nodes) stores the aggregate file size of all byte sizes of the nodes below it. So the roots cumulatively will describe all possible byte ranges for this repository.
- Find the root that contains the byte range of the offset you are looking for and get the node information for all of that nodes children using the Index Lookup method, and recursively repeat this step until you find the lowest down child node that describes this byte range.
- The chunk described by this child node will contain the byte range you are looking for. You can use the `byteOffset` property in the `Stat` metadata object to seek into the right position in the content for the start of this chunk.

Metadata Overhead

Using this scheme, if you write 4GB of data using on average 64KB data chunks (note: chunks can be variable length and do not need to be the same size), your tree file will be around 5MB (0.0125% overhead).

signatures

A SLEEP formatted 32 byte header with data entries being 64 byte signatures.

```
<32 byte header>
  <4 byte magic string: 0x05025701>
  <1 byte version number: 0>
  <2 byte entry size: 64>
  <1 byte algorithm name length prefix: 7>
  <7 byte algorithm name: Ed25519>
  <17 zeroes>
<64 byte entries>
  <64 byte Ed25519 signature>
```

Every time the tree is updated we sign the current roots of the Merkle tree, and append them to the signatures file. The signatures file starts with no entries. Each time a new leaf is appended to the `tree` file (aka whenever data is added to a Dat), we take all root hashes at the current state of the Merkle tree and hash and sign them, then append them as a new entry to the signatures file.

```
Ed25519 sign(
  BLAKE2b(
    <1 byte>
      2 // root type
    for (every root node left-to-right) {
      <32 byte root hash>
      <8 byte Uint64BE root tree index>
      <8 byte Uint64BE child byte lengths>
    }
  )
)
```

The reason we hash all the root nodes is that the BLAKE2b hash above is only calculateable if you have all of the pieces of data required to generate all the intermediate hashes. This is the crux of Dat's data integrity guarantees.

bitfield

A SLEEP formatted 32 byte header followed by a series of 3328 byte long entries.

```
<32 byte header>
  <4 byte magic string: 0x05025700>
  <1 byte version number: 0>
  <2 byte entry size: 3328>
  <1 byte algorithm name length: 0>
  <1 byte algorithm name: 0>
  <24 zeroes>
<3328 byte entries> // (2048 + 1024 + 256)
```

The bitfield describes which pieces of data you have, and which nodes in the `tree` file have been written. This file exists as an index of the `tree` and `data` to quickly figure out which pieces of data you have or are missing. This file can be regenerated if you delete it, so it is considered a materialized index.

The `bitfield` file actually contains three bitfields of different sizes. A bitfield (AKA bitmap) is defined as a set of bits where each bit (0 or 1) represents if you have or do not have a piece of data at that bit index. So if there is a dataset of 10 cat pictures, and you have pictures 1, 3, and 5 but are missing the rest, your bitfield would look like `1010100000`.

Each entry contains three objects:

- Data Bitfield (1024 bytes) - 1 bit for for each data entry that you have synced (1 for every entry in `data`).
- Tree Bitfield (2048 bytes) - 1 bit for every tree entry (all nodes in `tree`)
- Bitfield Index (256 bytes) - This is an index of the Data Bitfield that makes it efficient to figure out which pieces of data are missing from the Data Bitfield without having to do a linear scan.

The Data Bitfield is 1Kb somewhat arbitrarily, but the idea is that because most filesystems work in 4Kb chunk sizes, we can fit the Data, Tree and Index in less then 4Kb of data for efficient writes to the filesystem. The Tree and Index sizes are based on the Data size (the Tree has twice the entries as the Data, odd and even nodes vs just even nodes in `tree`, and Index is always 1/4th the size).

To generate the Index, you pairs of 2 bytes at a time from the Data Bitfield, check if all bites in the 2 bytes are the same, and generate 4 bits of Index metadata for every 2 bytes of Data (hence how 1024 bytes of Data ends up as 256 bytes of Index).

First you generate a 2 bit tuple for the 2 bytes of Data:

```
if (data is all 1's) then [1,1]
if (data is all 0's) then [0,0]
if (data is not all the same) then [1, 0]
```

The Index itself is an in-order binary tree, not a traditional bitfield. To generate the tree, you take the tuples you generate above and then write them into a tree like the following example, where non-leaf nodes are generated using the above scheme by looking at the results of the relative even child tuples for each odd parent tuple:

```
// for e.g. 16 bytes (8 tuples) of
// sparsely replicated data
0 - [00 00 00 00]
1 -     [10 10 10 10]
2 - [11 11 11 11]
```

The tuples at entry `1` above are `[1,0]` because the relative child tuples are not uniform. In the following example, all non-leaf nodes are `[1,1]` because their relative children are all uniform (`[1,1]`)

```
// for e.g. 32 bytes (16 tuples) of
// fully replicated data (all 1's)
0 - [11 11 11 11]
1 -     [11 11 11 11]
2 - [11 11 11 11]
3 -         [11 11 11 11]
4 - [11 11 11 11]
5 -     [11 11 11 11]
6 - [11 11 11 11]
```

Using this scheme, to represent 32 bytes of data it takes at most 8 bytes of Index. In this example it compresses nicely as its all contiguous ones on disk, similarly for an empty bitfield it would be all zeroes.

If you write 4GB of data using on average 64KB data chunk size, your bitfield will be at most 32KB.

metadata.data

This file is used to store content described by the rest of the `metadata.*` hypercore SLEEP files. Whereas the `content.*` SLEEP files describe the data stored in the actual data cloned in the Dat repository filesystem, the `metadata` data feed is stored inside the `.dat` folder along with the rest of the SLEEP files.

The contents of this file is a series of versions of the Dat filesystem tree. As this is a hypercore data feed, it's just an append only log of binary data entries. The challenge is representing a tree in an one dimensional way to make it representable as a Hypercore register. For example, imagine three files:

```
~/dataset $ ls
figures
  graph1.png
  graph2.png
results.csv

1 directory, 3 files
```

We want to take this structure and map it to a serialized representation that gets written into an append only log in a way that still allows for efficient random access by file path.

To do this, we convert the filesystem metadata into entries in a feed like this:

```
{
  "path": "/results.csv",
  children: [[]],
  sequence: 0
}
{
  "path": "/figures/graph1.png",
  children: [[0], []],
  sequence: 1
}
{
  "path": "/figures/graph2",
  children: [[0], [1]],
  sequence: 2
}
```

Filename Resolution

Each sequence represents adding one of the files to the register, so at sequence 0 the filesystem state only has a single file, `results.csv` in it. At sequence 1, there are only 2 files added to the register, and at sequence 3 all files are finally added. The `children` field represents a shorthand way of declaring which other files at every level of the directory hierarchy exist alongside the file being added at that revision. For example at the time of sequence 1, children is `[[0], []]`. The first sub-array, `[0]`, represents the first folder in the `path`, which is the root folder `/`. In this case `[0]` means the root folder at this point in time only has a single file, the file that is the subject of sequence `0`. The second subarray is empty `[]` because there are no other existing files in the second folder in the `path`, `figures`.

To look up a file by filename, you fetch the latest entry in the log, then use the `children` metadata in that entry to look up the longest common ancestor based on the parent folders of the filename you are querying. You can then recursively repeat this operation until you find the `path` entry you are looking for (or you exhaust all options which means the file does not exist). This is a `O(number of slashes in your path)` operation.

For example, if you wanted to look up `/results.csv` given the above register, you would start by grabbing the metadata at sequence 2. The longest common ancestor between `/results.csv` and `/figures/graph2` is `/`. You then grab the corresponding entry in the children array for `/`, which in this case is the first entry, `[0]`. You then repeat this with all of the chilren entries until you find a child that is closer to the entry you are looking for. In this example, the first entry happens to be the match we are looking for.

You can also perform lookups relative to a point in time by starting from a specific sequence number in the register. For example to get the state of some file relative to an old sequence number, similar to checking out an old version of a repository in Git.

Data Serialization

The format of the `metadata.data` file is as follows:

```
<Header>
<Node 1>
<Node 2>
<Node ...>
<Node N>
```

Each entry in the file is encoded using Protocol Buffers (Varda 2008).

The first message we write to the file is of a type called Header which uses this schema:

```
message Header {
  required string type = 1;
  optional bytes content = 2;
}
```

This is used to declare two pieces of metadata used by Dat. It includes a `type` string with the value `hyperdrive` and `content` binary value that holds the public key of the content register that this metadata register represents. When you share a Dat, the metadata key is the main key that gets used, and the content register key is linked from here in the metadata.

After the header the file will contain many filesystem `Node` entries:

```
message Node {
  required string path = 1;
  optional Stat value = 2;
  optional bytes children = 3;
}
```

The `Node` object has three fields

- `path` - the string of the absolute file path of this file.
- `Stat` - a Stat encoded object representing the file metadata
- `children` - a compressed list of the sequence numbers as described earlier

The `children` value is encoded by starting with the nested array of sequence numbers, e.g. `[[3], [2, 1]]`. You then sort the individual arrays, in this case resulting in `[[3], [1, 2]]`. You then delta compress each subarray by storing the difference between each integer. In this case it would be `[[3], [1, 1]]` because `3` is 3 more than 0, `1` is 1 more than than 0, and `2` is 1 more than `1`.

When we write these delta compressed subarrays we write them using variable width integers (varints), using a repeating pattern like this, one for each array:

```
<varint of first subarray element length>
<varint of the first delta in this array>
<varint of the next delta ...>
<varint of the last delta>
```

This encoding is designed for efficiency as it reduces the filesystem path metadata down to a series of small integers.

The `Stat` objects use this encoding:

```
message Stat {
  required uint32 mode = 1;
  optional uint32 uid = 2;
  optional uint32 gid = 3;
  optional uint64 size = 4;
  optional uint64 blocks = 5;
  optional uint64 offset = 6;
  optional uint64 byteOffset = 7;
  optional uint64 mtime = 8;
  optional uint64 ctime = 9;
}
```

These are the field defintions:

- `mode` - posix file mode bitmask
- `uid` - posix user id
- `gid` - posix group id
- `size` - file size in bytes
- `blocks` - number of data chunks that make up this file
- `offset` - the data feed entry index for the first chunk in this file
- `byteOffset` - the data feed file byte offset for the first chunk in this file
- `mtime` - posix modified_at time
- `mtime` - posix created_at time

4. 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>`.

Register

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

Handshake

Type 1. Overall connection handshake. Should be sent just after the register 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

```
message Handshake {
  optional bytes id = 1;
  optional bool live = 2;
}
```

Status

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 Status {
  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 an 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 it's 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;
  }
```

```
}
```

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 (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.

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 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.

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, 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".

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*.

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

Varda, Kenton. 2008. "Protocol Buffers: Google's Data Interchange Format." *Google Open Source Blog, Available at Least as Early as Jul*.