SLEEP - Syncable Ledger of Exact Events Protocol

Mathias Buus Madsen, Maxwell Ogden, Code for Science

August 2017

SLEEP

This document 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 designed to allow efficient access to subsets of the metadata 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 Ledger of Exact Events Protocol. The Syncable 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, 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 children's 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 calculable 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 take pairs of 2 bytes at a time from the Data Bitfield, check if all bits 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 a 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",
  trie: [[]],
  sequence: 0
}
{
  "path": "/figures/graph1.png",
  trie: [[0], []],
  sequence: 1
}
{
  "path": "/figures/graph2.png",
  trie: [[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 children 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 trie = 3;
  repeated Writer writers = 4;
  optional uint64 writersSequence = 5;
}

message Writer {
  required bytes publicKey = 1;
  optional string permission = 2;
}
```

The `Node` object has five fields

- `path` - the string of the absolute file path of this file.
- `Stat` - a Stat encoded object representing the file metadata
- `trie` - a compressed list of the sequence numbers as described earlier
- `writers` - a list of the writers who are allowed to write to this dat
- `writersSequence` - a reference to the last sequence where the writers array was modified. you can use this to quickly find the value of the writers keys.

The `trie` value is encoded by starting with the nested array of sequence numbers, e.g. `[[[0, 3]], [[0, 2], [0, 1]]]`. Each entry is a tuple where the first item is the index of the feed in the `writers` array and the second value is the sequence number. Finally you prepend the trie value with a version number varint.

To write these subarrays we use variable width integers (varints), using a repeating pattern like this, one for each array:

```
<varint of 0>
<varint of 3>
<varint of 0>
<varint of 2>
<varint of 0>
<varint of 1>
```

This encoding is designed for efficiency as it reduces the filesystem path + feed index 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 definitions:

- `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

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