STRATO

HiDrive
Synchronization

Table of contents

1 Introduction

The HiDrive API provides features to enable data synchronization between one or more local clients that use HiDrive as remote storage. Synchronization can be achieved in a three-stage process.

The startup stage begins when the sync client establishes a connection to the server. Once connected, the client subscribes to change notifications and then compares the server's remote state with its own local state, building a synchronization plan. All server-side changes are then replicated to local data if applicable; conflicts may occur in this stage and need to be handled according to the conflict resolution rules.

In the second stage, the client replicates all remaining local changes to the server.

Once the replication of changes in both directions has completed successfully, the client can enter an operational mode, the third stage. The server will notify all subscribed clients whenever server-side changes to the file system occur.

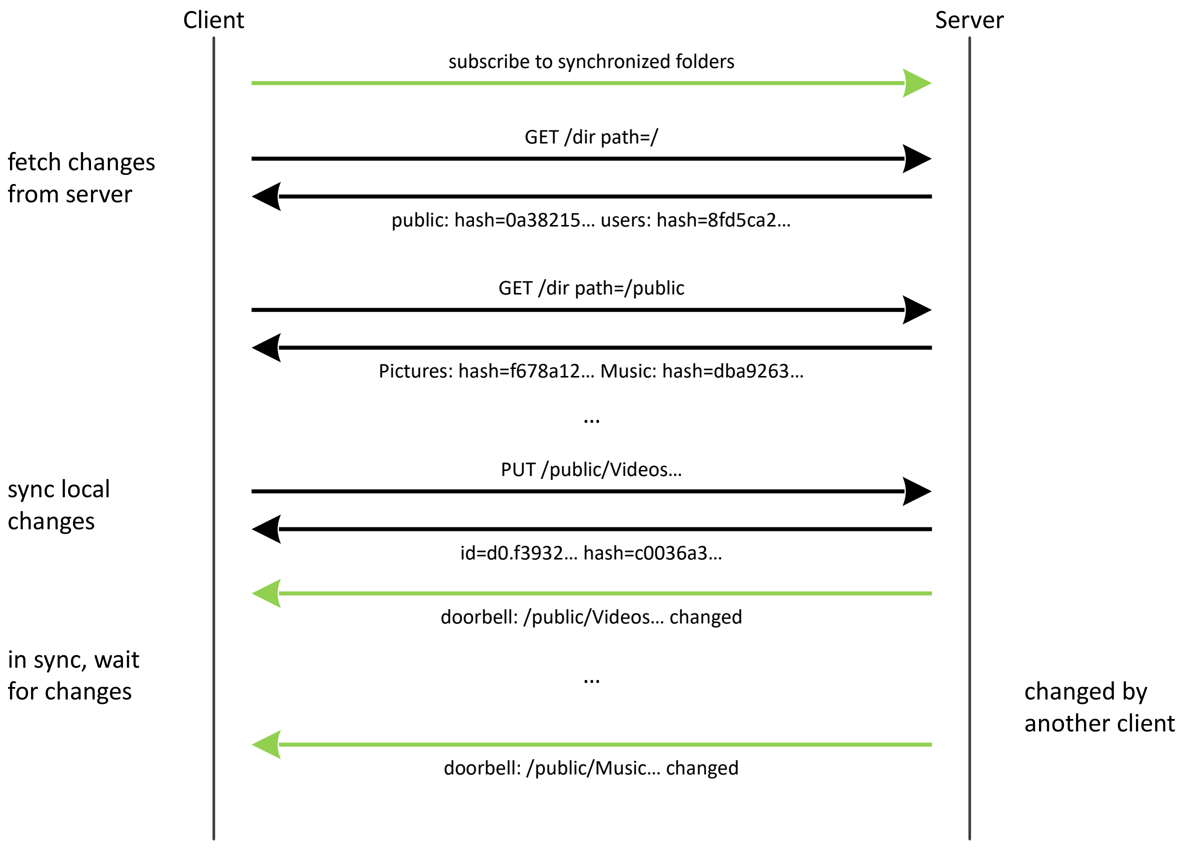
Client
Server
subscribe to synchronized folders
fetch changes from server
GET /dir path=/
public: hash=0a38215... users: hash=8fd5ca2...
GET /dir path=/public
Pictures: hash=f678a12... Music: hash=dba9263...
...
sync local changes
PUT /public/Videos...
id=d0.f3932... hash=c0036a3...
doorbell: /public/Videos... changed
in sync, wait for changes
...
changed by another client
doorbell: /public/Music... changed

Figure 1: Overview (black arrows: API communication, green: change notifications via websocket)

Change detection is based on hash values that are calculated for both content and metadata (name and last modification time and, for files, the size). For file content data, the smallest unit is a hash calculated for a 4096-byte block. Hashes for content and metadata can be combined so that a single hash value represents the entire state of a sub-tree.

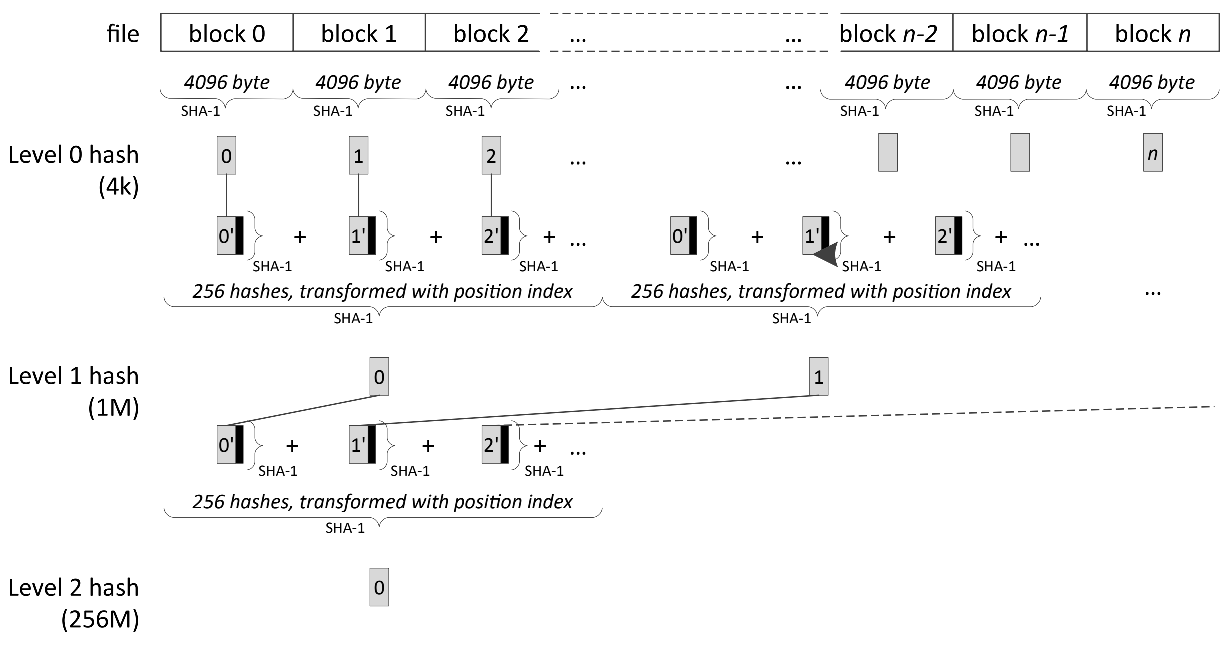

Figure 2: File content hashing overview 4096 bytes

2 Startup mode

Server-to-Client replication

To replicate the remote server's state to the local state, the client can request a listing of the top-level directory that contains hash information. By comparing the remote directory hashes to locally generated hashes, the client can identify modifications. If the hashes for a directory are the same, the content in that directory does not need to be examined, because everything contained in and below these directories is the same on both sides. A recursive comparison is required if the hashes differ.

File content can be compared using the top-level content hash of the file. If the hashes differ, sub-level hashes can be requested from the server to identify blocks that have diverged from the local copy from the top down. This way, the client can identify and selectively download only those remote chunks that are sufficient to replicate the server-side state.

Some operations can have a huge and unnecessary impact if not handled properly. For example, consider a rename operation on the remote side. In a naïve interpretation, this appears to be a deletion followed by creation of a new entry. Replicating these steps faithfully would result in an unnecessary transfer of data already present on the local system. Similarly, the amount of allocated quota on HiDrive would double if a rename operation on the local side is processed as a new upload followed by a deletion of the original file.

Note that a rename operation on the remote side for a directory or file results in a new file system entry with new metadata hashes, but its content hash value remains the same and is the same as an already known, local entry. It is advisable to build a list of all operations which that can be identified in this way, such as *rename*, *copy*, and *move*. Detecting these operations on the file level is required and while the same optimization is possible for the directory structure, it is usually not necessary.

Client-to-Server replication

It is assumed that a client represents the HiDrive data using a virtual file system[1] and thus is "aware" of the local state at all times. Local changes that occurred since the last sync can simply be replicated to the remote server unless they require conflict resolution.

3 Operational mode

Change notification — "doorbells"

The client can subscribe to a websocket-based change notification service for one or more directories. Once subscribed for a directory, the client will receive doorbell events whenever the subscribed directory itself or, recursively, any file system entries in the subtree are modified. Please refer to the "[subs2-doorbell documentation]" for details.

A client should subscribe before entering startup mode so that changes occurring during the initial synchronization phase can subsequently be replicated.

When the client receives a notification, it compares the remote hashes with the local version to detect and locate changed entries and then requests the updated file, or chunks thereof, from the server.

Write operations

While the client is connected to the server, any write operation to a local entry should be directly replicated to HiDrive. Shortly thereafter, the server's doorbell event will confirm the operation and the remote hashes can be checked against the local version to verify.

Caching

The client should offer local caching for non-synchronized directories, both for metadata (file name, timestamps, file size, etc.) and data on a block level.

For example, if the user opens a directory in Windows Explorer, the local metadata cache is validated against the server using the metadata-only hashes. Also, the client should start a subscription for that directory to enable a near real-time display of changed entries and to update or invalidate cached data where required.

When a file is opened locally, the available cached data blocks are validated against the server. Invalid cached data blocks are discarded. When uncached data is accessed, it is fetched from the server, placed into the cache and then returned to the application. Additionally, some data can be prefetched to enable buffering for streaming applications.

[1] Example implementations include FUSE for Unix-based systems and EldoS Callback File System, Dokan or FileSystemWatcher for Windows.

Conflict resolution

Conflict resolution is only required during the initial startup phase. When the client has achieved a synced state and enters the operational phase, any local changes should be directly replicated to HiDrive as described in the section *Write operations*.

During the initial startup phase, the client fetches hashes for changed data from HiDrive. These new remote hashes are then compared to the ones dating from the last successful in-sync state or to local hashes before a local change was made. If the hashes are identical, the local change can be uploaded to HiDrive directly, otherwise the local change and the remote state are in conflict. To resolve this conflict, the local file first needs to be locally renamed and uploaded to HiDrive. Thereafter, the file is recovered locally from the remote version.

4 Hashes in detail

Introduction

One of the requirements for an efficient synchronization protocol is the ability to determine which data is not in sync. Support for change detection and data comparison in HiDrive is built on hashes. While having a single hash for a file's content is viable for small files, it becomes more time-consuming to calculate the hash as the file size increases. For that reason, HiDrive offers hierarchical content hashes on multiple levels combined with metadata hashes that allow a client to pinpoint a change down to a 4096-byte block of content data as well as to verify efficiently whether two whole directory subtrees are identical.

A sync client that reconnects to HiDrive can request a single 20-byte value from the API and, through a comparison with a locally generated value, find out if there were any changes on the remote side. If the hash values match, the client is still in sync. Otherwise, it can descend into the directory tree to identify changed directories and files.

Via the HiDrive API, hashes can be requested for:

- File content (hierarchical) and metadata (name, size and mtime)
- Directories; name, data they contain, metadata and subdirectories (i.e. recursive)
- Metadata of directories (non-recursive)

For these hashes to be useful during the synchronization, a sync client must be able to generate these hashes locally as well.

File content

Content hashes (chash) for a file are structured hierarchically: First, for each 4096-byte sized block of raw file content, the respective SHA-1 digest[2] is calculated. This is called a Level 0 hash (L_0). No hash is calculated (an "empty" hash) if the file content block contains only null bytes or belongs to a hole in a sparse file (see below for details). If the last block of a file is shorter than 4096 bytes, the block's content data is padded with null bytes before calculating the hash value.

A Level 1 hash can be formed by aggregating 256 of the L_0 hashes after a transformation to include the block's position: Each 4096-byte block is assigned a position index *i* [0,255] that is encoded as an 8-bit value. The index byte is appended to the 20 bytes of each L_0 hash and a new SHA-1 digest over <L_0-hash$_i$, index$_i$> is calculated. If a content block contained only null bytes and thus no corresponding L_0 hash was calculated (the hash at the index position is empty), the index position is simply incremented and processing continues with

[2] https://tools.ietf.org/html/rfc3174

the next L_0 hash. All transformed level 0 hashes are then added[3] modulo 2^{160}, producing a 20-byte long Level 1 hash covering one megabyte of content data.

This aggregation function—append index byte to L_n hash, generate SHA-1 digest, add digests to obtain the L_{n+1} hash—can be applied repeatedly: 256 L_1 hashes can be aggregated to form a Level 2 hash L_2 covering 256M, 256 L_2 hashes can be transformed into a L_3 hash (64G) and so on.

The "top-level hash" is reached when a single 20-byte hash value remains from the last aggregation step and all other hashes on the level are considered empty. This 20-byte top-level hash corresponds to the entirety of the file content: if even a single byte is changed in the content, the avalanche property of the SHA-1 digest algorithm will, with overwhelming probability, lead to a different top-level hash. The level of the top-level hash for a given file is determined by the file's size: a file of up to 4 kilobytes in size requires a single L_0 hash, files up to one megabyte in size can be covered by a L_1 hash, and a L_2 hash is sufficient for a 256 MB file and so on.

Please refer to the hash calculation example in the appendix for details.

Sparse or empty file optimization

This hierarchical hashing algorithm does not differentiate between longer, contiguous sequences of null bytes and "holes" in files, so called sparse files. At the lowest level L_0, no hashes (empty hash slots) are generated for blocks that contain only null bytes. When, during the aggregation step, all 256 lower-level hash slots are found to be empty, the higher-level hash slot will also be empty. That way, large holes in sparse files can be represented efficiently.
If the file consists *only* of null bytes or is a single sparse hole, there are no hashes on any level, regardless of the file size. In this case, the file content hash is represented by a special hash value consisting of 20 null bytes[4], as the top-level hash cannot be empty.

Rationale for this file content hashing algorithm

As described, a single changed byte in the middle of a large file will trickle up as changes in the hierarchy of hashes and finally lead to a different top-level hash. When compared to its earlier state, this file can readily be identified as changed based on a single comparison of the old and new top-level hashes. Assuming that the sync client has a local database of file content hashes, the location where a remote change occurred can be tracked down by descending through the hash levels to the affected block. Also note that, once the updated L_0 hash for the changed block has been calculated, all higher hash levels can be re-calculated locally without having to read the rest of the file. Once the changed block is uploaded to the server, the file's server-side top-level hash should confirm the locally calculated new top-level hash.

Clients operating with limited storage space may optimize hash calculation by storing hashes only for level 1 and up and generating L_0 hashes on the fly.

Combining hashes and retrieving server-side hashes

File content hashes — chash

Content hashes for files can be retrieved with the `GET /file/hash` API call; multiple byte ranges for a hash level can be specified upon request. The response will always include the top-level file content hash `chash`.

The top-level `chash` for files can also be requested by adding the `members.chash` value to the `fields` parameter for a `GET /dir` or `GET /meta` API call.

[3] Addition modulo 2^{160} refers to a byte-wise addition of two positive 160-bit integers, starting at the least significant byte and with carry of the arithmetic overflow. After the last addition, a possibly present carry bit is discarded so that the result always fits into 160 bits.
[4] 0x0000000000000000000000000000000000000000

File system entry names — nhash

Name hashes (nhash) help to avoid issues when different encoding schemes are used at the file system level and, as a consequence, a client's local representation of the name differs from the server-side interpretation. A client operating with a different encoding can resolve the file system entry name by mapping a local name to a remote nhash to identify the remote name.

The hash of the name of a file system entry for a file or directory is the SHA-1 hash of the name as it is stored server-side.

The name hash can be requested by adding the members.nhash value to the fields parameter for a GET /dir call or by adding nhash to the list of fields parameters to the GET /meta call. When a file system entry is created server-side as the result of an API call, the name hash will always be included in the response.

Metadata hashes — mhash

The metadata hash mhash for a file is calculated by hashing the following components: the SHA-1 hash of the filename hash (nhash), followed by the file's size as a 64-bit little endian integer and the UNIX timestamp of the last modification time, also represented as a 64-bit little endian integer:

$$\text{mhash}_f = \text{sha1}(\text{nhash}_f, \text{le64}(\text{size}_f), \text{le64}(\text{mtime}_f))$$

For a directory, the size is omitted from the metadata hash:

$$\text{mhash}_d = \text{sha1}(\text{nhash}_d, \text{le64}(\text{mtime}_d))$$

Directory hash (including content, recursive)

The relative top-level hash of a directory is its content hash chash. It is created via the addition, again modulo 2^{160}, of the metadata hashes and the content hashes for files and, if present, the mhash and chash values of the subdirectories contained.

$$\text{chash}_d = \text{mhash}_{f1} + \text{chash}_{f1} + \text{mhash}_{f2} + \text{chash}_{f2} + \ldots + \text{mhash}_{d1} + \text{chash}_{d1} + \text{mhash}_{d2} + \text{chash}_{d2}$$

By including the metadata and content hashes of subdirectories in a directory's content hash, the hash value recursively covers the complete sub-tree.

Directory hash (metadata only, non-recursive)

The metadata-only hash mohash for a directory is generated in the same way as the directory content hash, except that all content hashes are ignored:

$$\text{mohash}_d = \text{mhash}_{f1} + \text{mhash}_{f2} + \ldots$$

The metadata-only hash for a directory is non-recursive. It is relevant for use in caching because the content of files and directories may not be available locally.

Change notification service

HiDrive offers a change notification service via the websocket protocol where a connected client can subscribe to one or more directories in order to receive a "doorbell" event whenever the top-level hashes for a subscribed directory change. The event data includes the new mhash and chash. The rationale is that a client no longer needs to poll in a loop (pull) but receives notifications only when changes occur (push), reducing the workload of the client.

5 Hash calculation examples

The content of the example file is created[5] using a 64-byte string (including a newline character at the end) as an elementary unit:

`#ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz\n`

A 4096-byte long sequence can be constructed by concatenating 64 of these units; this is a Level 0 block. This block of data can then be "covered" by calculating a Level 0 hash as the 20-byte SHA-1 digest of the block's content. The hexadecimal representation of the L_0 hash for this Level 0 block is:

```
MSByte    20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0  LSByte
             09 f0 77 82 0a 8a 41 f3 4a 63 9f 21 72 f1 13 3b 1e af e4 e6
```

The first megabyte of the example file's content is created by repeating this block until 256 blocks have been written. This first megabyte can be covered by a Level 1 checksum. To calculate the Level 1 checksum, the L_0 hashes need to be transformed. First, the position of the block (modulo 256) is appended to each L_0 hash as the new least significant byte:

```
L0 Block  20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0  LSByte
       0  09 f0 77 82 0a 8a 41 f3 4a 63 9f 21 72 f1 13 3b 1e af e4 e6 00
       1  09 f0 77 82 0a 8a 41 f3 4a 63 9f 21 72 f1 13 3b 1e af e4 e6 01
       2  09 f0 77 82 0a 8a 41 f3 4a 63 9f 21 72 f1 13 3b 1e af e4 e6 02
   … 255  09 f0 77 82 0a 8a 41 f3 4a 63 9f 21 72 f1 13 3b 1e af e4 e6 ff
```

Next, SHA-1 hashes over these transformed L_0 hashes inputs are calculated:

```
L0 Block  20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0  LSByte
       0     44 fe 5c a6 34 25 68 b4 16 7b f9 90 b6 4e 40 4a 39 75 e1 c3
       1     4b d3 99 be 7d b3 43 31 3c 95 62 f6 8e 14 0b fe 9f a2 81 ed
       2     96 02 09 99 61 99 bd 68 93 0e 4d cb f4 f3 19 d4 43 30 6f 6b
   … 255     12 9b fe d9 d8 c9 20 6e dd 7d ae 8c 21 44 05 9a ce 57 ce bf
```

Finally, the 256 hashes resulting from these transformations are added, starting at the least significant byte and with carry. The addition of the two most significant bytes may result in a value to be carried but it is discarded, thus making the addition effectively modulo 2^{160}. Adding the transformed hashes for block 0 and 1 shown in detail:

```
   19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
   44  fe  5c  a6  34  25  68  b4  16  7b  f9  90  b6  4e  40  4a  39  75  e1  c3
   4b  d3  99  be  7d  b3  43  31  3c  95  62  f6  8e  14  0b  fe  9f  a2  81  ed
   68 254  92 166  52  37 104 180  22 123 249 144 182  78  64  74  57 117 225 195
+  75 211 153 190 125 179  67  49  60 149  98 246 142  20  11 254 159 162 129 237
C  +1      +1                      +1  +1  +1  +1          +1      +1  +1  +1
  144 465 246 356 177 216 171 229  83 273 348 391 324  98  76 328 217 280 355 432
M   %   %   %   %   %       %           %   %       %           %   %   %   %   %
  144 209 246 100 177 216 171 229  83  17  92 135  68  98  76  72 217  24  99 176
S  90  d1  f6  64  b1  d8  ab  e5  53  11  5c  87  44  62  4c  48  d9  18  63  b0
```

To the sum resulting from adding the hashes for block 0 and 1, the hash of block 2 can be added:

[5] We provide a script to generate the sample file along with other sample code available at <URL>

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	90	d1	f6	64	b1	d8	ab	e5	53	11	5c	87	44	62	4c	48	d9	18	63	b0
+	96	02	09	99	61	99	bd	68	93	0e	4d	cb	f4	f3	19	d4	43	30	6f	6b
	26	**d3**	**ff**	**fe**	**13**	**72**	**69**	**4d**	**e6**	**1f**	**aa**	**53**	**39**	**55**	**66**	**1d**	**1c**	**48**	**d3**	**1b**

The addition of all 256 transformed L_0 hashes yields the resulting L_1 hash that covers the first megabyte (block 0 on Level 1):

L1 *Block*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*LSByte*
0		**75**	**a9**	**f8**	**8f**	**b2**	**19**	**ef**	**1d**	**d3**	**1a**	**df**	**41**	**c9**	**3e**	**2e**	**fa**	**ac**	**8d**	**02**	**45**	

In the example file, the second megabyte data block begins with 128 blocks of the 4096-byte example sequence while the remaining 128 blocks are empty. Depending on the file system used, the result is either a sparse file with a hole for that range, or null bytes are actually written out to disk.

To calculate the second L_1 hash, the hashes for the L_0 blocks are summed up in exactly the same way as was done for the first L_1 block. After block 128, the remaining data blocks are empty (or contain null bytes). By definition, an empty block is simply skipped during L_0 hash calculation and thus, after adding up the transformed L_0 hashes for blocks 256 to 384[6], the sum remains unchanged for the rest of the blocks, resulting in this L_1 hash for the second megabyte:

L1 *Block*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*LSByte*
1		da	ed	c4	25	19	95	01	b1	e8	6b	5e	ab	a5	64	9c	bd	e2	05	e6	ae	

The example file ends with two complete 4096-byte example sequences followed by the first 2048 bytes of the example sequence. As the last L_0 block is incomplete, it has to be padded with null bytes before calculating the L_0 SHA-1 hashes. The resulting L_0 hashes with appended relative block positions are:

L0 *Block*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*LSByte*
512	09	f0	77	82	0a	8a	41	f3	4a	63	9f	21	72	f1	13	3b	1e	af	e4	e6	**00**	
513	09	f0	77	82	0a	8a	41	f3	4a	63	9f	21	72	f1	13	3b	1e	af	e4	e6	**01**	
514	fd	cf	d1	8f	27	7c	6f	82	0d	c8	b8	51	e3	c8	57	d8	86	3b	97	ff	**02**	

Calculating SHA-1 hashes over these inputs yields:

L0 *Block*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*LSByte*
512		44	fe	5c	a6	34	25	68	b4	16	7b	f9	90	b6	4e	40	4a	39	75	e1	c3	
513		4b	d3	99	be	7d	b3	43	31	3c	95	62	f6	8e	14	0b	fe	9f	a2	81	ed	
514		97	98	ce	c3	8d	c1	18	fb	9e	05	27	08	c5	db	ed	1d	43	1e	fa	26	

Adding the three transformed L_0 hashes for blocks 512-514 modulo 2^{160} directly results in the L_1 hash because the file ends after the third block and the remaining hashes are all skipped:

L1 *Block*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*LSByte*
2		28	6a	c5	28	3f	99	c4	e0	f1	16	83	90	0a	3e	39	66	1c	37	5d	d6	

Now the three Level 1 hashes for the example file can be transformed to include the block position, hashed again and then added to form a Level 2 hash:

L1 *Block*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*LSByte*
0	75	a9	f8	8f	b2	19	ef	1d	d3	1a	df	41	c9	3e	2e	fa	ac	8d	02	45	**00**	
1	da	ed	c4	25	19	95	01	b1	e8	6b	5e	ab	a5	64	9c	bd	e2	05	e6	ae	**01**	
2	28	6a	c5	28	3f	99	c4	e0	f1	16	83	90	0a	3e	39	66	1c	37	5d	d6	**02**	

[6] Absolute block positions "256 to 384" are the relative (modulo 256) block positions 0 to 127 in the second megabyte of the file.

Hashed:

```
L1 Block   20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0  LSByte
     T0       ad 7b 84 f5 b0 ac 2b b7 79 28 42 fc 65 f9 bc c1 a0 bd 02 74
     T1       28 22 08 ee 25 05 ed b8 4f 69 0a a7 ab fe 0d e5 84 70 fe 08
     T2       27 70 1a 56 be 23 64 65 1c 83 7a bf fe 90 ef 6b fd 7c 68 cb
```

Added modulo 2^{160}, the resulting L_2 hash—in this case the content hash chash that covers the entire example file—is:

```
L2 Block      19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0  LSByte
      0       fd 0d a8 3a 93 d5 7d d4 e5 14 c8 64 10 88 ba 13 22 aa 69 47
```

These calculations can be verified using the API call /file/hash on the example file with parameters level=1 and range=-. The response is:

```
{
    "chash": "fd0da83a93d57dd4e514c8641088ba1322aa6947",
    "level": 2,
    "list": [
        [
            {
                "block": 0,
                "hash": "75a9f88fb219ef1dd31adf41c93e2efaac8d0245",
                "level": 1
            },
            {
                "block": 1,
                "hash": "daedc425199501b1e86b5eaba5649cbde205e6ae",
                "level": 1
            },
            {
                "block": 2,
                "hash": "286ac5283f99c4e0f11683900a3e39661c375dd6",
                "level": 1
            }
        ]
    ]
}
```

Name and metadata hashes: nhash and mhash

Assume a directory that is named "HiDrive ☁", i.e. the ASCII letters HiDrive followed by a space and the Unicode character "CLOUD" (U+2601), or in hexadecimal notation:

```
        H    i    D    r    i    v    e         ☁
Hex    48   69   44   72   69   76   65   20   e2 98 81
```

The corresponding name hash nhash is generated by applying SHA-1 to the raw bytes of the name. Note that API responses contain the name in URL encoding and hash values are represented in hexadecimal notation:

```
{
    "name": "HiDrive%20%E2%98%81",
    "nhash": "f72f99f62d1142f67ac32be03043c0c2adb3ab88"
}
```

To generate the metadata mhash for a directory, the UNIX timestamp of the last modification time mtime is also required; in this example it is 1456789012 or 2016-02-29T23:36:52+00:00 in ISO-8610 format (UTC).

To calculate the mhash, the UNIX timestamp 1456789012 is first converted into a 64-bit little endian integer resulting in eight bytes 14d6d45600000000 (hex). The 20 bytes of the nhash and the eight bytes from the converted timestamp are concatenated and the mhash is the result of applying SHA-1 to this input. The result is shown here:

```
{
    "name": "HiDrive%20%E2%98%81",
    "nhash": "f72f99f62d1142f67ac32be03043c0c2adb3ab88",
    "mtime": 1456789012,
    "mhash": "4f450fa02257ea368179557f482e73b2fb80b566"
}
```

Note that the mtime can become negative: if the last modification time is set to an hour before the UNIX epoch, the mtime would be -3600 or f0f1ffffffffffff (hex). The mhash for the same directory name is:

```
{
    "name": "HiDrive%20%E2%98%81",
    "nhash": "f72f99f62d1142f67ac32be03043c0c2adb3ab88",
    "mtime": -3600,
    "mhash": "a287b73ebad0c931c85f6a0e60af534f009d071f"
}
```

When the mhash is calculated for files, the file size is used as a 64-bit little endian integer in the input for the SHA-1 operation: first come the 20 bytes of the name hash followed by the eight bytes of the file size and finally the eight bytes of the last modification time.

The file used for this example is the same one previously used in the explanation of file content hashes; its last modification time was set to 1234567890 or 2009-02-13T23:31:30+00:00 (UTC).
The API response for this file is:

```
{
    "name": "sample.bin",
    "mtime": 1234567890,
    "size": 2107392,
    "chash": "fd0da83a93d57dd4e514c8641088ba1322aa6947",
    "mhash": "449fee596b27c879052e9d82366cb5d63ebaf6f6",
    "nhash": "7220d977d2db4499f333bfff421158b9815a686f"
}
```

And finally, a query for the directory “HiDrive ☁” containing only this sample file yields this response:

```
{
    "name": "HiDrive%20%E2%98%81",
    "nhash": "f72f99f62d1142f67ac32be03043c0c2adb3ab88",
    "chash": "41ad9693fefd464dea4365e646f56fe96165603d",
    "mtime": 1456789012,
    "mhash": "4f450fa02257ea368179557f482e73b2fb80b566",
    "mohash": "449fee596b27c879052e9d82366cb5d63ebaf6f6",
    "members": [
        {
            "name": "sample.bin",
            "nhash": "7220d977d2db4499f333bfff421158b9815a686f",
            "chash": "fd0da83a93d57dd4e514c8641088ba1322aa6947",
            "mtime": 1234567890,
            "size": 2107392,
            "mhash": "449fee596b27c879052e9d82366cb5d63ebaf6f6"
        }
    ]
}
```

The metadata-only hash for the directory `mohash` is in this case identical to the `mhash` of the file because only that file contributes a metadata hash to the `chash` and the `mohash` of the directory itself.