# FoxFS specification

Moritz "LittleFox" Grosch

September 29, 2015

## Contents



# 1 Preface

## 1.1 Why another filesystem?

This filesystem has to be implemented in JavaScript, so it has to be simple. No journaling, no access control lists, no unix permissions, no owner ... and so on.

Using something like FAT wasn't an option, because of filesize-limits. Just using 64bit attributes makes FAT very inefficient. We need a better filesystem ... so I started working on FoxFS and got it to a state worth specifying after just 3 days. It is not yet completed and please don't trust it to store your valuable data, but please test it and report errors so I can fix them.

## 1.2 Design

FoxFS has a layered design: below FoxFS is the block-access layer provided by the device-driver.

On top of this is the blockchain-layer, this is the first layer in FoxFS and implements data-storage inside blockchains (not the Bitcoin-ones, but just a chain of blocks e.g. first load block 5, then 7 and then 6).

On the foundation of the blockchain-layer, FoxFS builds the filesystem-layer, this is where the actual filesystem is implemented. In the filesystem-layer we have directories, files, file-attributes, ... everything you would expect from a filesystem.

# 2 Datastructures in FoxFS

## 2.1 inode

An inode is a simple structure, which points to a data-block and the following inode. It is used to build the blockchains, where we put all the data. The format is shown in table 1.

Table 1: Format of an inode

| **Field** | **Datatype** | **Description** |
|---|---|---|
| `block_idx` | unsigned 64bit integer | The block containing the data for this inode |
| `next_inode_block_idx` | unsigned 64bit integer | The block containing the next inode |
| `next_inode_offset` | unsigned 32bit integer | The offset of the inode in the block, given in bytes |
| `reserved` | 12 bytes | Has to be set to 0 all the time |

The data associated with the inode can be found in the block referenced by `block_idx`. The next inode can be found with the `next_inode_*`-fields, if there is no following inode, these are set to 0.

To get all data in a blockchain, read the block referenced by the first inode, get the next inode, load the block referenced in that inode, get the next inode, ... until there is no following inode.

Inodes are identified by the block where they are stored and the offset from the begin-

ning of the block given as inode-index.

There are some special inodes, these are described in table 2.

Table 2: Special inodes

| Block | Offset | Usage |
|---|---|---|
| 0 | 0 | Points to the blockchain of the root-directory |
| 0 | 1 | Points to the blockchain of the free-space bitmapt |
| 0 | 2 | Points to the inode-blockchain |
| 0 | 2 | Points to the blockchain for the filesystem-metadata directory |

## 2.2 Free-space bitmap

It's a bitmap with a flag for every block in the filesystem. It just tells us if the block is available (= 1) or unavailable (= 0). It can be used to find a free block to store new data.

A block can be unavailable for many reasons, for example damage or it is just already used. In either way, it is always just marked as unavailable, it's not our department to track disk-health.

## 2.3 Directories

This is the core-function of every filesystem: mapping filenames and metadata to actual filecontent.

Directories in FoxFS are stored in a blockchain. Actually, there isn't any difference between directories and files. Directories are just files with a fileformat known by the filesystem-driver.

The format of directories is specified in table 3. All entries are saved without holes in the structure, making the files less fragmented and less space-consuming. This way, we only need to save the number of entries and not their position or a valid flag in every entry. If a driver deletes an entry, it has to fill the hole with other entries. The order of the entries is guaranted not to be preserved.

As you can see in table 4, a directory-entry only contains references to inodes. There is one inode for the data-blockchain, which contains the actual file content and another inode for metadata-blockchain.

With this design, we can store an arbitrary amount of metadata in the filesystem, much

more than the normal filename, size and different timestamps - we could store changelogs or diffs there for example.

Table 3: Structure of a directory

| Field | Datatype | Description |
|---|---|---|
| num_entries | unsigned 64bit integer | The number of entries in this directory |
| children_size | unsigned 64bit integer | Last known size of all children |
| entries | array of structure defined in table 4 | The actual entries in this directory |

### 2.4 File-metadata

This is the structure stored in the meta-blockchain and referenced by every file.

## 3 The layers of FoxFS

### 3.1 Block-layer

This is actually not a layer from FoxFS, but the layer below it. FoxFS needs just three functions in this layer: storeBlock, loadBlock and getNumBlocks - everything else is implemented on these. FoxFS also supports the trimBlock function, enabling the block-layer to know which blocks are free.

### 3.2 Blockchain-layer

This is the first layer in FoxFS. It implements chains of blocks to store data in. Everything in FoxFS is stored in such a blockchain.

Blockchains are identified by the inode pointing to their first block.

Table 4: Structure of an fileentry

| Field | Datatype | Description |
|---|---|---|
| content_inode_block_idx | unsigned 64bit integer | Block containing the inode for the content-blockchain |
| content_inode_offset | unsigned 32bit integer | Offset of the inode in the block given in content_inode_block_idx |
| meta_inode_block_idx | unsigned 64bit integer | Block containing the inode for the metadata-blockchain |
| meta_inode_offset | unsigned 32bit integer | Offset of the inode in the block given in meta_inode_block_idx |
| filename | unsigned 8bit integer[30] | Name of the file. If the name is longer, it is stored in the metadata and this field is filled with zeros. The name is stored as UTF-8 and without trailing zero. |
| size | unsigned 64bit integer | Size of the file given in bytes |
| attributes | unsigned 16bit integer | some attributes:<br>• 0x01 Entry is a directory<br>• 0x02 Entry is a file |
| reserved | unsigned 8bit integer | reserved for future use. Set to 0 on write and keep value on update. |

Table 5: Metadata structure

| Field | Datatype | Description |
|---|---|---|
| magic | unsigned 32bit integer | 'M3TA' |
| num_entries | unsigned 64bit integer | Number of metadata-entries |