Obnam2—a backup system

Lars Wirzenius

2022-05-21 04:29

Contents

Chapter 1

Abstract

Obnam is a backup system, consisting of a not very smart server for storing chunks of backup data, and a client that splits the user's data into chunks. They communicate via HTTP.

This document describes the architecture and acceptance criteria for Obnam, as well as how the acceptance criteria are verified.

Chapter 2

Introduction

Obnam2 is a backup system.

In 2004 I started a project to develop a backup program for myself, which in 2006 I named Obnam. In 2017 I retired the project, because it was no longer fun. The project had some long-standing, architectural issues related to performance that had become entrenched and were hard to fix, without breaking backwards compatibility.

In 2020, with Obnam2 I'm starting over from scratch. The new software is not, and will not become, compatible with Obnam1 in any way. I aim the new software to be more reliable and faster than Obnam1, without sacrificing security or ease of use, while being maintainable in the long run. I also intend to have fun while developing the new software.

Part of that maintainability is going to be achieved by using Rust as the programming language (which has a strong, static type system) rather than Python (which has a dynamic, comparatively weak type system). Another part is more strongly aiming for simplicity and elegance. Obnam1 used an elegant, but not very simple copy-on-write B-tree structure; Obnam2 will use SQLite[1].

2.1 Glossary

This document uses some specific terminology related to backups. Here is a glossary of such terms.

- a **chunk** is a relatively small amount of live data or metadata about live data, as chosen by the client
- a **client** is the computer system where the live data lives, also the part of Obnam running on that computer

[1] `https://sqlite.org/`

- a **generation** is a snapshot of live data, also known as **a backup**
- **live data** is the data that gets backed up
- a **repository** is where the backups get stored
- a **server** is the computer system where the repository resides, also the part of Obnam running on that computer

Chapter 3

Requirements

The following high-level requirements are not meant to be verifiable in an automated way:

- *Done:* **Easy to install:** available as a Debian package in an APT repository. Other installation packages will also be provided, hopefully.
- *Ongoing:* **Easy to configure:** only need to configure things that are inherently specific to a client, when sensible defaults are impossible.
- *Not done:* **Excellent documentation:** although software ideally does not need documentation, in practice is usually does, and Obnam should have documentation that is clear, correct, helpful, unambiguous, and well-liked.
- *Done*: **Easy to run:** making a backup is a single command line that's always the same.
- *Ongoing:* **Detects corruption:** if a file in the repository is modified or deleted, the software notices it automatically.
- *Ongoing:* **Repository is encrypted:** all data stored in the repository is encrypted with a key known only to the client.
- *Ongoing:* **Fast backups and restores:** when a client and server both have sufficient CPU, RAM, and disk bandwidth, the software makes a backup or restores a backup over a gigabit Ethernet using at least 50% of the network bandwidth.
- *Done:* **Snapshots:** Each backup is an independent snapshot: it can be deleted without affecting any other snapshot.
- *Done:* **Deduplication:** Identical chunks of data are stored only once in the backup repository.
  - Note: The chunking is very simplistic, for now, but that can be improved later. The changes will only affect the backup part of the client.
- *Not done:* **Compressed:** Data stored in the backup repository is compressed.
- *Not done:* **Large numbers of live data files:** The system must handle

at least ten million files of live data. (Preferably much more, but I want some concrete number to start with.)

- *Not done:* **Live data in the terabyte range:** The system must handle a terabyte of live data. (Again, preferably more.)
- *Not done:* **Many clients:** The system must handle a thousand total clients and one hundred clients using the server concurrently, on one physical server.
- *Not done:* **Shared repository:** The system should allow people who don't trust each other to share a repository without fearing that their own data leaks, or even its existence leaks, to anyone.
- *Not done:* **Shared backups:** People who do trust each other should be able to share backed up data in the repository.
- *Done:* **Limited local cache:** The Obnam client may cache data from the server locally, but the cache should be small, and its size must not be proportional to the amount of live data or the amount of data on the server.
- *Not done*: **Resilient:** If the metadata about a backup or the backed up data is corrupted or goes missing, everything that can be restored must be possible to restore, and the backup repository must be possible to be repaired so that it's internally consistent.
- *Not done*: **Self-compatible:** It must be possible to use any version of the client with any version of the backup repository, and to restore any backup with any later version of the client.
- *Not done:* **No re-backups:** The system must never require the user to do more than one full backup the same repository.

The detailed, automatically verified acceptance criteria are documented below, as *scenarios* described for the Subplot[1] tool. The scenarios describe specific sequences of events and the expected outcomes.

[1] `https://subplot.liw.fi/`

Chapter 4

Threat model

This chapter discusses the various threats against backups. Or it will. For now it's very much work in progress. This version of the chapter is only meant to get threat modeling started by having the simplest possible model that is in any way useful.

4.1 Backed up data is readable by server operator

This threat is about the operator of the backup server being able to read the data backed up by any user of the server. We have to assume that the operator can read any file and can also eavesdrop all network traffic. The operator can even read all physical and virtual memory on the server.

The mitigation strategy is to encrypt the data before it is sent to the server. If the server never receives cleartext data, the operator can't read it.

Backups have four kinds of data:

- actual contents of live data files
- metadata about live data files, as stored on the client file system, such as the name, ownership, or size of each file
- metadata about the contents of live data, such as its cryptographic checksum
- metadata about the backup itself

For now, we are concerned about the first two kinds. The rest will be addressed later.

The mitigation technique against this threat is to encrypt the live data and its metadata before uploading it to the server.

4.2 An attacker with access to live data can stealthily exclude files from the backup

This threat arises from Obnam's support for CACHEDIR.TAG[1] files. As the spec itself says in the "Security Considerations" section:

> "Blind" use of cache directory tags in automatic system backups could potentially increase the damage that intruders or malware could cause to a system. A user or system administrator might be substantially less likely to notice the malicious insertion of a CACHDIR.TAG into an important directory than the outright deletion of that directory, for example, causing the contents of that directory to be omitted from regular backups.

This is mitigated in two ways:

1. if an incremental backup finds a tag which wasn't in the previous backup, Obnam will show the path to the tag, and exit with a non-zero exit code. That way, the user has a chance to notice the new tag. The backup itself is still made, so if the tag is legitimate, the user doesn't need to re-run Obnam.

   Error messages and non-zero exit are jarring, so this approach is not user-friendly. Better than nothing though;

2. users can set `exclude_cache_tag_directories` to `false`, which will make Obnam ignore the tags, nullifying the threat.

   This is a last-ditch solution, since it makes the backups larger and slower (because Obnam has to back up more data).

[1] `https://bford.info/cachedir/`

Chapter 5

Software architecture

5.1 Effects of requirements

The requirements stated above drive the software architecture of Obnam. Some requirements don't affect the architecture at all: for example, "excellent documentation". This section discusses the various requirements and notes how they affect the architecture.

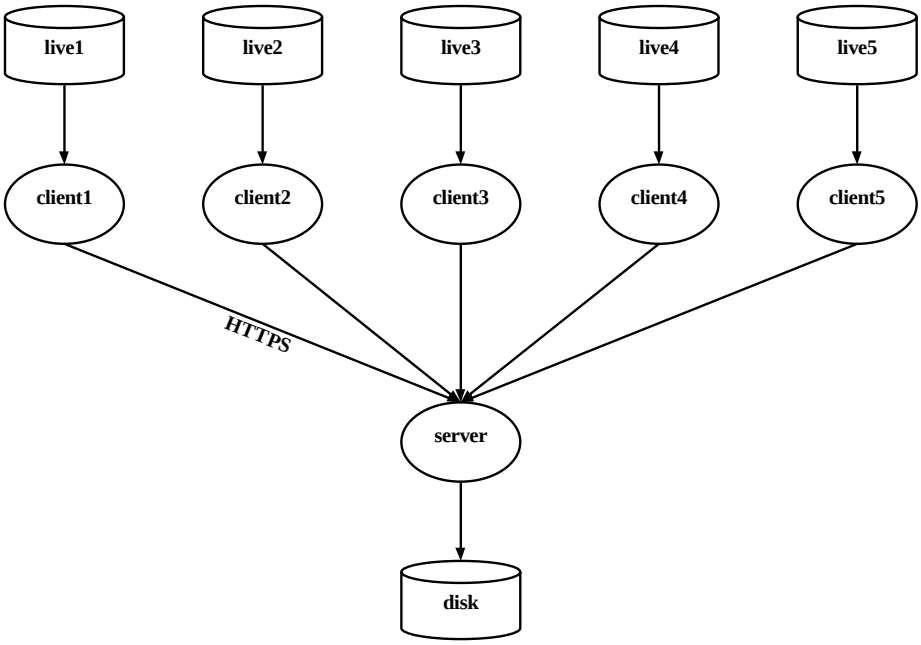
- **Easy to install:** Does not affect the architeture.
- **Easy to configure:** Does not affect the architeture.
- **Excellent documentation:** Does not affect the architeture.
- **Easy to run:** Obnam may not require that its user provide any information specific to a backup run. For example, it may not require a name or identifier to be provided. The software must invent any identifiers itself.
- **Detects corruption:** The client must provide a strong checksum of the data it uploads, and verify the checksum for data it downloads. Note that the server can't compute or verify the checksum, at least not for the cleartext data, which it never sees. Also, having the server compute a checksum is too late: corruption may have happened during the upload already.
- **Repository is encrypted:** Client must do the encryption and decryption. The server may only see encrypted data. Note that this must include metadata, such as the checksum of cleartext data. The client will encrypt the checksum for a chunk and the server must not interpret or use the checksum in any way.
- **Fast backups and restores:** The architecture needs to enable the implementation to use concurrency and protocols that can saturate fast network connections, and handle network problems well.
- **Snapshots:** We can't do deltas from one backup run to another. If Obnam does a tape-like full backup, and then an incremental one as a delta from

the full one, it can't delete the full backup until all the incremental ones have been deleted. This complicated management of backup storage.

- **Deduplication:** The client sees the cleartext and can make more intelligent decisions about how to split live data into chunks. Further, the client has fast access to the live data, which the server does not. Ideally, we design the server in a way that does not care about how data is split into chunks.
- **Compressed:** Compression should be done prior to encryption: if encrypted data can be significantly compressed that leaks information about the nature of the cleartext data.
- **Large numbers of live data files:** Storing and accessing lists of and meta data about files needs to done using data structures that are efficient for that.
- **Live data in the terabyte range:** FIXME
- **Many clients:** The architecture should enable flexibly managing clients.
- **Shared repository:** The server component needs identify and distinguish between clients and data in backups made by different clients. Access to backups to be strictly controlled so that each client can only ever access its own data, or even query about the presence of specific data.
- **Shared backups:** Clients should be able to specify, for each chunk of data separately, which other clients should be able to access that.

5.2 Overall shape

It seems fairly clear that a simple shape of the software architecture of Obnam2 is to have a client and server component, where one server can handle any number of clients. They communicate over HTTPS, using proven web technologies for authentication and authorization.



The responsibilities of the server are roughly:

- provide an HTTP API for managing chunks and their metadata: create, retrieve, search, delete; note that updating a chunk is not needed
- keep track of the client owning each chunk
- allow clients to manage sharing of specific chunks between clients

The responsibilities of the client are roughly:

- split live data into chunks, upload them to server
- store metadata of live data files in a generation file (an SQLite database), store that too as chunks on the server
- retrieve chunks from server when restoring
- let user manage sharing of backups with other clients

There are many details to add to both to the client and the server, but that will come later.

It is possible that an identity provider needs to be added to the architecture later, to provide strong authentication of clients. However, that will not be necessary for the minimum viable product version of Obnam. For the MVP, authentication will happen using RSA-signed JSON Web Tokens. The server is configured to trust specific public keys. The clients have the private keys and generate the tokens themselves.

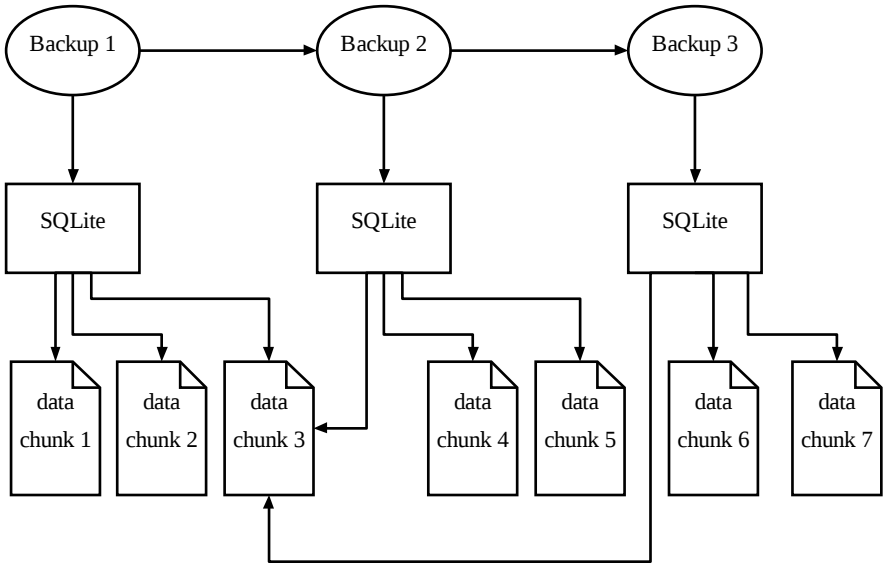
5.3 Logical structure of backups

For each backup (generation) the client stores, on the server, exactly one *generation chunk*. This is a chunk that is specially marked as a generation, but is otherwise not special. The generation chunk content is a list of identifiers for chunks that form an SQLite database.

The SQLite database lists all the files in the backup, as well as their metadata. For each file, a list of chunk identifiers are listed, for the content of the file. The chunks may be shared between files in the same backup or different backups.

File content data chunks are just blobs of data with no structure. They have no reference to other data chunks, or to files or backups. This makes it easier to share them between files.

Let's look at an example. In the figure below there are three backups, each using three chunks for file content data. One chunk, "data chunk 3", is shared between all three backups.



5.4 Evolving the database

The per-generation SQLite database file has a schema. Over time it may be necessary to change the schema. This needs to be done carefully to avoid having backup clients to have to do a full backup of previously backed up data.

We do this by storing the "schema version" in the database. Each database will have a table `meta`:

```
CREATE TABLE meta (key TEXT, value TEXT)
```

This will allow key/value pairs serialized into text. We use the keys `schema_version_major` and `schema_version_minor` to store the schema version. This will allow the Obnam client to correctly restore the backup, or at least do the best job it can, while warning the user there may be an incompatibility.

We may later add more keys to the `meta` table if there's a need.

The client will support every historical major version, and the latest historical minor version of each major version. We will make sure that this will be enough to restore every previously made backup. That is, every backup with schema version x.y will be possible to correctly restore with a version of the Obnam client that understands schema version x.z, where $z >= y$. If we make a change that would break this, we increment the major version.

We may drop support for a schema version if we're sure no backups with that schema version exist. This is primarily to be able to drop schema versions that were never included in a released version of the Obnam client.

To verify schema compatibility support, we will, at minimum, have tests that automatically make backups with every supported major version, and restore them.

5.5 On SFTP versus HTTPS

Obnam1 supported using a standard SFTP server as a backup repository, and this was a popular feature. This section argues against supporting SFTP in Obnam2.

The performance requirement for network use means favoring protocols such as HTTPS, or even QUIC, rather than SFTP.

SFTP works on top of SSH. SSH provides a TCP-like abstraction for SFTP, and thus multiple SFTP connections can run over the same SSH connection. However, SSH itself uses a single TCP connection. If that TCP connection has a dropped packet, all traffic over the SSH connections, including all SFTP connections, waits until TCP re-transmits the lost packet and re-synchronizes itself.

With multiple HTTP connections, each on its own TCP connection, a single dropped packet will not affect other HTTP transactions. Even better, the new QUIC protocol doesn't use TCP.

The modern Internet is to a large degree designed for massive use of the world wide web, which is all HTTP, and adopting QUIC. It seems wise for Obnam to make use of technologies that have been designed for, and proven to work well with concurrency and network problems.

Further, having used SFTP with Obnam1, it is not always an easy protocol to

use. Further, if there is a desire to have controlled sharing of parts of one client's data with another, this would require writing a custom SFTP service, which seems much harder to do than writing a custom HTTP service. From experience, a custom HTTP service is easy to do. A custom SFTP service would need to shoehorn the abstractions it needs into something that looks more or less like a Unix file system.

The benefit of using SFTP would be that a standard SFTP service could be used, if partial data sharing between clients is not needed. This would simplify deployment and operations for many. However, it doesn't seem important enough to warrant the implementation effort.

Supporting both HTTP and SFTP would be possible, but also much more work and against the desire to keep things simple.

5.6 On "btrfs send" and similar constructs

The btrfs and ZFS file systems, and possibly others, have a way to mark specific states of the file system and efficiently generate a "delta file" of all the changes between the states. The delta can be transferred elsewhere, and applied to a copy of the file system. This can be quite efficient, but Obnam won't be built on top of such a system.

On the one hand, it would force the use of specific file systems: Obnam would no be able to back up data on, say, an ext4 file system, which seems to be the most popular one by far.

Worse, it also for the data to be restored to the same type of file system as where the live data was originally. This onerous for people to do.

5.7 On content addressable storage

It would be possible to use the cryptographic checksum ("hash") of the contents of a chunk as its identifier on the server side, also known as content-addressable storage[1]. This would simplify de-duplication of chunks. However, it also has some drawbacks:

- it becomes harder to handle checksum collisions
- changing the checksum algorithm becomes harder

In 2005, the author of git version control system[2] chose the content addressable storage model, using the SHA1 checksum algorithm. At the time, the git author considered SHA1 to be reasonably strong from a cryptographic and security point of view, for git. In other words, given the output of SHA1, it was difficult to deduce what the input was, or to find another input that would give the same

[1] `https://en.wikipedia.org/wiki/Content-addressable_storage`
[2] `https://git-scm.com/`

output, known as a checksum collision. It is still difficult to deduce the input, but manufacturing collisions is now feasible, with some constraints. The git project has spent years changing the checksum algorithm.

Collisions are problematic for security applications of checksum algorithms in general. Checksums are used, for example, in storing and verifying passwords: the cleartext password is never stored, and instead a checksum of it is computed and stored. To verify a later login attempt a new checksum is computed from the newly entered password from the attempt. If the checksums match, the password is accepted.[3] This means that if an attacker can find *any* input that gives the same output for the checksum algorithm used for password storage, they can log in as if they were a valid user, whether the password they have is the same as the real one.

For backups, and version control systems, collisions cause a different problem: they can prevent the correct content from being stored. If two files (or chunks) have the same checksum, only one will be stored. If the files have different content, this is a problem. A backup system should guard against this possibility.

As an extreme and rare, but real, case consider a researcher of checksum algorithms. They've spent enormous effort to produce two distinct files that have the same checksum. They should be able make a backup of the files, and restore them, and not lose one. They should not have to know that their backup system uses the same checksum algorithm they are researching, and have to guard against the backup system getting the files confused. (Backup systems should be boring and just always work.)

Attacks on security-sensitive cryptographic algorithms only get stronger by time. It is therefore necessary for Obnam to be able to easily change the checksum algorithm it uses, without disruption for user. To achieve this, Obnam does not use content-addressable storage.

Obnam will (eventually, as this hasn't been implemented yet) allow storing multiple checksums for each chunk. It will use the strongest checksum available for a chunk. Over time, the checksums for chunks can be replaced with stronger ones. This will allow Obnam to migrate to a stronger algorithm when attacks against the current one become too scary.

5.8 On pull versus push backups

Obnam only does push backups. This means the client runs on the host where the live data is, and sends it to the server.

Backups could also be pulled, in that the server reaches out tot he host where the live data is, retrieves the data, and stores it on the server. Obnam does not do this, due to the hard requirement that live data never leaves its host in cleartext.

[3] In reality, storing passwords securely is much more complicated than described here.

The reason pull backups are of interest in many use cases is because they allow central administration of backups, which can simplify things a lot in a large organization. Central backup administration can be achieved with Obnam in a more complicated way: the installation and configuration of live data hosts is done in a central fashion using configuration management, and if necessary, backups can be triggered on each host by having the server reach out and run the Obnam client.

Chapter 6

File metadata

Files in a file system contain data and have metadata: data about the file itself. The most obvious metadata is the file name, but there is much more. A backup system needs to back up, but also restore, all relevant metadata. This chapter discusses all the metadata the Obnam authors know about, and how they understand it, and how Obnam handles it, and why it handles it that way.

The long term goal is for Obnam to handle everything, but it may take a while to get there.

6.1 On portability

Currently, Obnam is developed on Linux, and targets Linux only. Later, it may be useful to add support for other systems, and Obnam should handle file metadata in a portable way, when that makes sense and is possible. This means that if a backup is made on one type of system, but restored on another type, Obnam should do its best to make the restored data as identical as possible to what the data would be if it had been copied over directly, with minimal change in meaning.

This affects not only cases when the operating system changes, but also when the file system changes. Backing up on Linux ext4 file system and restoring to a vfat file system brings up the same class of issues with file metadata.

There are many type of file systems[1] with varying capabilities and behaviors. Obnam attempts to handle everything the Linux system it runs on can handle.

[1] `https://en.wikipedia.org/wiki/Comparison_of_file_systems`

6.2 Filenames

On Unix, the filename is a sequence of bytes. Certain bytes have special meaning:

- byte 0, ASCII NUL character: terminates filename
- byte 56, ASCII period character: used for . and .. directory entries
- byte 57, ASCII slash character: used to separate components in a pathname

On generic Unix, the operating system does not interpret other bytes. It does not impose a character set. Binary filenames are OK, as long as they use the above bytes only in the reserved manner. It is up to the presentation layer (the user interface) to present the name in a way suitable for humans.

For now, Obnam stores fully qualified pathnames as strings of bytes as above. Arguably, Obnam could split the pathname into components, stored separately, to avoid having to give ASCII slash characters special meaning. The `.` and `..` directory entries are not stored by Obnam.

Different versions of Unix, and different file system types, put limits on the length of a filename or components of a pathname. Obnam does not.

On other operating systems, and on some file system types, filenames are more restricted. For example, on MacOS, although nominally a Unix variant, filenames must form valid UTF-8 strings normalized in a particular way. While Obnam does not support MacOS at the time of writing, if it ever will, that needn't affect the way filenames are stored. They will be stored as strings of bytes, and if necessary, upon restore, a filename can be morphed into a form required by MacOS or the filename being written to. The part of Obnam that restores files will have to learn how to do that.

The generic Unix approach does not allow for "drive letters", used by Windows. Not sure if supporting that is needed.

6.3 Unix inode metadata: `struct stat`

The basic Unix system call for querying a file's metadata is stat(2)[2]. However, since it follows symbolic links, Obnam needs to use lstat(2)[3] instead. The metadata is stored in an inode[4]. Both variants return a C `struct stat`. On Linux, it has the following fields:

- `st_dev` — id of the block device containing file system where the file is; this encodes the major and minor device numbers
  - this field can't be restored as such, it is forced by the operating system for the file system to which files are restored
  - Obnam stores it so that hard links can be restored, see below
- `st_ino` — the inode number for the file

[2] `https://linux.die.net/man/2/stat`
[3] `https://linux.die.net/man/2/lstat`
[4] `https://en.wikipedia.org/wiki/Inode`

  - this field can't be restored as such, it is forced by the file system whan the restored file is created
  - Obnam stores it so that hard links can be restored, see below
- `st_nlink` — number of hard links referring to the inode
  - this field can't be restored as such, it is maintained by the operating system when hard links are created
  - Obnam stores it so that hard links can be restored, see below
- `st_mode` — file type and permissions
  - stored and restored
- `st_uid` — the numeric id of the user account owning the file
  - stored
  - restored if restore is running as root, otherwise not restored
- `st_gid` — the numeric id of the group owning the file
  - stored
  - restored if restore is running as root, otherwise not restored
- `st_rdev` — the device this inode represents
  - not stored
- `st_size` — size or length of the file in bytes
  - stored
  - restored implicitly be re-creating the origtinal contents
- `st_blksize` — preferred block size for efficient I/O
  - chosen automatically by the operating system, can't be changed
  - not stored
- `st_blocks` — how many blocks of 512 bytes are actually allocated to store this file's contents
  - see below for discussion about sparse files
  - not stored
- `st_atime` — timestamp of latest access
  - stored and restored
  - On Linux, split into two integer fields to achieve nanosecond resolution
- `st_mtime` — timestamp of latest modification
  - stored and restored
  - On Linux, split into two integer fields to achieve nanosecond resolution
- `st_ctime` — timestamp of latest inode change
  - can't be set by an application, maintained automatically by operating system
  - not stored

Obnam stores most of these fields. Not all of them can be restored, especially not explicitly. The `st_dev` and `st_ino` fields get set by the file system when when a restored file is created. They're stored so that Obnam can restore all hard links to the same inode.

6.4 Hard links and symbolic links

In Unix, filenames are links to an inode. The inode contains all the metadata, except the filename. Many names can link to the same inode. These are called hard links.

On Linux, hard links can be created explicitly only for regular files, not for directories. This avoids creating cycles in the directory tree, which simplifies all software that traverses the file system. However, hard links get created implicitly when creating sub-directories: the `..` entry in the sub-directory is a hard link to the inode of the parent directory.

Unix also supports symbolic links, which are tiny files that contain the name of another file. The kernel will follow a symbolic link automatically by reading the tiny file, and pretending the contents of the file was used instead. Obnam stores the contents of a symbolic link, the "target" of the link, and restores the original value without modification.

To recognize that filename are hard links to the same file, a program needs to use **lstat**(2) on each filename and compare the `st_dev` and `st_ino` fields of the result. If they're identical, the filenames refer to the same inode. It's important to check both fields so that one is certain the resulting data refers to the same inode on the same file system. Keeping track of filenames pointing at the same inode can be resource intensive. It can be helpful to note that it only needs to be done for inodes with an `st_nlink` count greater then one.

6.5 On access time stamps

The `st_atime` field is automatically updated when a file or directory is "accessed". This means reading a file or listing the contents of a directory. Accessing a file in a directory does count as accessing the directory.

The `st_atime` update can be prevented by updating the file system as read-only, or using a mount option `noatime`, `nodiratime`, or `relatime`, or by opening the file or directory with the `O_NOATIME` option (under certain conditions). This can be a useful for a system administrator to do to avoid needless updates if nothing needs the access timestamp. There are few uses for it.

Strictly speaking, a backup program can't assume the access timestamp is not needed and should do its best to back it up and restore it. However, this is trickier that one might think. A backup program can't change mount options, or make the file system be read-only. It thus needs to use the `NO_ATIME` flag to the open(2)[5] system call.

Obnam does not do this yet. In fact, it doesn't store or restore the access time stamp, and it might never do that. If you have a need for that, please open issue

[5] `https://linux.die.net/man/2/open`

on the Obnam issue tracker[6].

6.6 Time stamp representation

Originally, Unix (and Linux) stored file time stamps as whole seconds since the beginning of 1970. Linux now stores file timestamps with up to nanosecond precision, depending on file system type. Obnam handles this by storing and restoring nanosecond timestamps. If, when restoring, the target file system doesn't support that precision, then some accuracy is lost.

Different types of file system store timestamps at different precision, and sometimes support a different precision for different types of timestamp. The Linux ext4[7] file system supports nanosecond precision for all timestamps. The FAT[8] file system supports a 2 seconds for last modified time, 10 ms for creation time, 1 day for access date (if at all), 2 seconds for deletion time.

Obnam uses the same Linux system calls for retrieve timestamps, and those always return them at nanosecond precision (if not accuracy). Likewise when restoring, Obnam attempts to set the timestamps in the same way, and if the target file system supports less precision, the result may be imperfect, but there isn't really anything Obnam can do to improve that

6.7 Sparse files

On Unix a sparse file[9] is one where some blocks of the file are not stored explicitly, but the file still has a length. Instead, the file system return zero bytes for the missing blocks. The blocks that aren't explicitly stored form "holes" in the file.

As an example, one can create a very large file with the command line truncate(1)[10] command:

```
$ truncate --size 1T sparse
$ ls -l sparse
-rw-rw-r-- 1 liw liw 1099511627776 Dec  8 11:18 sparse
$ du sparse
0   sparse
```

It's a one-terabyte long file that uses no space! If the file is read, the file system serves one terabyte of zero bytes. If it's written, the file system creates a new block at the location of the write, and fills it new data, and fills the rest of the block with zeroes.

[6] https://gitlab.com/obnam/obnam/-/issues
[7] https://en.wikipedia.org/wiki/Ext4
[8] https://en.wikipedia.org/wiki/File_Allocation_Table
[9] https://en.wikipedia.org/wiki/Sparse_file
[10] https://linux.die.net/man/1/truncate

The metadata fields `st_size` and `st_blocks` make this visible. The `ls` command shows the `st_size` field. The `du` command reports disk usage based on the `st_blocks` field.

Sparse files are surprisingly useful. They can, for example, be used to implement large virtual disks without using more space than is actually stored on the file system on the virtual disk.

Sparse files are a challenge to backup systems: it is wasteful to store very large amounts of zeroes. Upon restore, the hole should be re-created rather then zeroes written out, or else the restored files will use much more disk space than the original files.

Obnam will store sparse files explicitly. It will find the holes in a file and store only the parts of a file that are not holes, and their position. But this isn't implemented yet.

6.8 Access control lists (Posix ACL)

FIXME

6.9 Extended attributes

FIXME

6.10 Extra Linux ext2/3/4 metadata

FIXME

6.11 On implementation and abstractions

Obnam clearly needs to abstract metadata across target systems. There are two basic appraches:

- every target gets its own, distinct metadata structure: LinuxMetadata, NetbsdMetadata, MacosMetadata, WindowsMetadata, and so on
- all targets share a common metadata structure that gets created in a target specific way

The first approach seems likely to cause an explosion of variants, and thus lead to more complexity overall. Thus, Obnam uses the second approach.

The Obnam source code has the `src/fsentry.rs` module, which is the common metadata structure, `FsEntry`. It has a default value that is adjusted using system specific functions, based on operating system specific variants of the `std::fs::Metadata` structure in the Rust standard library.

In addition to dealing with different `Metadata` on each system, the `FsEntry` needs to be stored in an SQLite database and retrieved from there. Initially, this will be done by serializing it into JSON and back. This is done at early development time, to simplify the process in which new metadata fields are added. It will be changed later, if there is need to.

Chapter 7

Implementation

The minimum viable product will not support sharing of data between clients.

7.1 Chunks

Chunks consist of arbitrary binary data, a small amount of metadata, and an identifier chosen by the server. The chunk metadata is a JSON object, consisting of the following field (there used to be more):

- `label` — the SHA256 checksum of the chunk contents as determined by the client
  - this MUST be set for every chunk, including generation chunks
  - the server allows for searching based on this field
  - note that the server doesn't verify this in any way, to pave way for future client-side encryption of the chunk data, including the label
  - there is no requirement that only one chunk has any given label

When creating or retrieving a chunk, its metadata is carried in a `Chunk-Meta` header as a JSON object, serialized into a textual form that can be put into HTTP headers.

There are several kinds of chunk. The kind only matters to the client, not to the server.

- Data chunk: File content data, from live data files, or from an SQLite database file listing all files in a backup.
- Generation chunk: A list of chunks for the SQLite file for a generation.
- Client trust: A list of ids of generation chunks, plus other data that are per-client, not per-backup.

7.2 Server

The server has the following API for managing chunks:

- `POST /v1/chunks` — store a new chunk (and its metadata) on the server, return its randomly chosen identifier
- `GET /v1/chunks/<ID>` — retrieve a chunk (and its metadata) from the server, given a chunk identifier
- `GET /v1/chunks?label=xyzzy` — find chunks on the server whose metadata has a specific value for a label.

HTTP status codes are used to indicate if a request succeeded or not, using the customary meanings.

When creating a chunk, chunk's metadata is sent in the `Chunk-Meta` header, and the contents in the request body. The new chunk gets a randomly assigned identifier, and if the request is successful, the response body is a JSON object with the identifier:

```
{
    "chunk_id": "fe20734b-edb3-432f-83c3-d35fe15969dd"
}
```

The identifier is a UUID4[1], but the client should not assume that and should treat it as an opaque value.

When a chunk is retrieved, the chunk metadata is returned in the `Chunk-Meta` header, and the contents in the response body.

It is not possible to update a chunk or its metadata. It's not possible to remove a chunk. When searching for chunks, any matching chunk's identifiers and metadata are returned in a JSON object:

```
{
  "fe20734b-edb3-432f-83c3-d35fe15969dd": {
     "label": "09ca7e4eaa6e8ae9c7d261167129184883644d07dfba7cbfbc4c8a2e08360d5b"
  }
}
```

There can be any number of chunks in the search response.

7.3 Client

The client scans live data for files, reads each file, splits it into chunks, and searches the server for chunks with the same checksum. If none are found, the client uploads the chunk. For each backup run, the client creates an SQLite[2] database in its own file, into which it inserts each file, its metadata, and list of

[1] `https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_(random)`
[2] `https://sqlite.org/`

chunk ids for its content. At the end of the backup, it uploads the SQLite file as chunks, and finally creates a generation chunk, which has as its contents the list of chunk identifiers for the SQLite file.

For an incremental backup, the client first retrieves the SQLite file for the previous generation, and compares each file's metadata with that of the previous generation. If a live data file does not seem to have changed, the client copies its metadata to the new SQLite file.

When restoring, the user provides the chunk id of the generation to be restored. The client retrieves the generation chunk, gets the list of chunk ids for the corresponding SQLite file, retrieves those, and then restores all the files in the SQLite database.

7.4 Encryption and authenticity of chunks

This is a plan that will be implemented soon. When it has been, this section needs to be updated to to use present tense.

Obnam encrypts data it stores on the server, and checks that the data it retrieves from the server is what it stored. This is all done in the client: the server should never see any data isn't encrypted, and the client can't trust the server to validate anything.

Obnam will be using *Authenticated Encryption with Associated Data* or AEAD[3]. AEAD both encrypts data, and validates it before decrypting. AEAD uses two encryption keys, one algorithm for symmetric encryption, and one algorithm for a message authentication codes or MAC[4]. AEAD encrypts the plaintext with a symmetric encryption algorithm using the first key, giving a ciphertext. It then computes a MAC of the ciphertext using the second key. Both the ciphertext and MAC are stored on the server.

For decryption, a MAC is computed against the retrieved ciphertext, and compared to the retrieved MAC. If the MACs differ, that's an error and no decryption is done. If they do match, the ciphertext is decrypted.

Obnam will require the user to provide a passphrase, and will derive the two keys from the single passphrase, using PBKDF2[5], rather than having the user provide two passphrases. The derived keys will be stored in a file that only the owner can read. (This is simple, and good enough for now, but needs to improved later.)

When this is all implemented, there will be a setup step before the first backup:

```
$ obnam init
Passphrase for encryption:
```

[3] `https://en.wikipedia.org/wiki/Authenticated_encryption#Authenticated_encryption_with_associated_data_(AEAD)`
[4] `https://en.wikipedia.org/wiki/Message_authentication_code`
[5] `https://en.wikipedia.org/wiki/PBKDF2`

```
Re-enter to make sure:
$ obnam backup
```

The `init` step asks for a passphrase, uses PBKDF2 (with the pbkdf2 crate[6]) to derive the two keys, and writes a JSON file with the keys into `~/.config/obnam/keys.json`, making that file be readable only by the user running Obnam. Other operations get the keys from that file. For now, we will use the default parameters of the pbkdf2 crate, to keep things simple. (This will need to be made more flexible later: if nothing else, Obnam should not be vulnerable to the defaults changing.)

The `init` step will not be optional. There will only be encrypted backups.

Obnam will use the aes-gcm crate[7] for AEAD, since it has been audited. If that choice turns out to be less than optimal, it can be reconsidered later. The `encrypt` function doesn't return the MAC and ciphertext separately, so we don't store them separately. However, each chunk needs its own nonce[8], which we will generate. We'll use a 96-bit (or 12-byte) nonce. We'll use the rand crate[9] to generate random bytes.

The chunk sent to the server will be encoded as follows:

- chunk format: a 32-bit unsigned integer, 0x0001, store in little-endian form.
- a 12-byte nonce unique to the chunk
- the ciphertext

The format version prefix dictates the content and structure of the chunk. This document defines version 1 of the format. The Obnam client will refuse to operate on backup generations which use chunk formats it cannot understand.

[6] `https://crates.io/crates/pbkdf2`
[7] `https://crates.io/crates/aes-gcm`
[8] `https://en.wikipedia.org/wiki/Cryptographic_nonce`
[9] `https://crates.io/crates/rand`

Chapter 8

Acceptance criteria for the chunk server

These scenarios verify that the chunk server works on its own. The scenarios start a fresh, empty chunk server, and do some operations on it, and verify the results, and finally terminate the server.

8.1 Chunk management happy path

We must be able to create a new chunk, retrieve it, find it via a search, and delete it. This is needed so the client can manage the storage of backed up data.

given a working Obnam system
and a file **data.dat** containing some random data
when I POST **data.dat** to **/v1/chunks**, with **chunk-meta**: **{"label":"0abc"}**
then HTTP status code is **201**
and **content-type** is **application/json**
and the JSON body has a field **chunk_id**, henceforth **ID**
and server has **1** chunks

We must be able to retrieve it.

when I GET /v1/chunks/<**ID**>
then HTTP status code is **200**
and **content-type** is **application/octet-stream**
and **chunk-meta** is **{"label":"0abc"}**
and the body matches file **data.dat**

We must also be able to find it based on metadata.

when I GET /v1/chunks?label=**0abc**
then HTTP status code is **200**

and **content-type** is **application/json**
and the JSON body matches **{"<ID>":{"label":"0abc"}}**

Finally, we must be able to delete it. After that, we must not be able to retrieve it, or find it using metadata.

when I DELETE /v1/chunks/<**ID**>
then HTTP status code is **200**
when I GET /v1/chunks/<**ID**>
then HTTP status code is **404**
when I GET /v1/chunks?label=**0abc**
then HTTP status code is **200**
and **content-type** is **application/json**
and the JSON body matches **{}**

8.2 Retrieve a chunk that does not exist

We must get the right error if we try to retrieve a chunk that does not exist.

given a working Obnam system
when I try to GET /v1/chunks/**any.random.string**
then HTTP status code is **404**

8.3 Search without matches

We must get an empty result if searching for chunks that don't exist.

given a working Obnam system
when I GET /v1/chunks?label=**0abc**
then HTTP status code is **200**
and **content-type** is **application/json**
and the JSON body matches **{}**

8.4 Delete chunk that does not exist

We must get the right error when deleting a chunk that doesn't exist.

given a working Obnam system
when I try to DELETE /v1/chunks/**any.random.string**
then HTTP status code is **404**

8.5 Persistent across restarts

Chunk storage, and the index of chunk metadata for searches, needs to be persistent across restarts. This scenario verifies it is so.

First, create a chunk.

given a working Obnam system
and a file **data.dat** containing some random data
when I POST **data.dat** to **/v1/chunks**, with **chunk-meta**: **{"label":"0abc"}**
then HTTP status code is **201**
and **content-type** is **application/json**
and the JSON body has a field **chunk_id**, henceforth **ID**

Then, restart the server.

when the chunk server is stopped
given a running chunk server

Can we still find it by its metadata?

when I GET /v1/chunks?label=**0abc**
then HTTP status code is **200**
and **content-type** is **application/json**
and the JSON body matches **{"<ID>":{"label":"0abc"}}**

Can we still retrieve it by its identifier?

when I GET /v1/chunks/<**ID**>
then HTTP status code is **200**
and **content-type** is **application/octet-stream**
and **chunk-meta** is **{"label":"0abc"}**
and the body matches file **data.dat**

8.6 Obeys `OBNAM_SERVER_LOG` environment variable

The chunk server logs its actions to stderr. Verbosity of the log depends on the `OBNAM_SERVER_LOG` envvar. This scenario verifies that the variable can make the server more chatty.

given a working Obnam system
and a file **data1.dat** containing some random data
when I POST **data1.dat** to **/v1/chunks**, with **chunk-meta**: **{"label":"qwerty"}**
then the JSON body has a field **chunk_id**, henceforth **ID**
and chunk server's stderr doesn't contain "**Obnam server starting up**"
and chunk server's stderr doesn't contain "**created chunk <ID>**"
given a running chunk server with environment **{"OBNAM_SERVER_LOG": "info"}**
and a file **data2.dat** containing some random data
when I POST **data2.dat** to **/v1/chunks**, with **chunk-meta**: **{"label":"xyz"}**
then the JSON body has a field **chunk_id**, henceforth **ID**

and chunk server's stderr contains "**Obnam server starting up**"
and chunk server's stderr contains "**created chunk <ID>**"

Chapter 9

Acceptance criteria for the Obnam client

The scenarios in chapter verify that the Obnam client works as it should, when it is used independently of an Obnam chunk server.

9.1 Client shows its configuration

This scenario verifies that the client can show its current configuration, with the `obnam config` command. The configuration is stored as YAML, but the command outputs JSON, to make sure it doesn't just copy the configuration file to the output.

given an installed obnam
and file **config.yaml**
and JSON file **config.json** converted from YAML file **config.yaml**
when I run **obnam --config config.yaml config**
then stdout, as JSON, has all the values in file **config.json**

File: **config.yaml**

```
roots: [live]
server_url: https://backup.example.com
verify_tls_cert: true
```

9.2 Client expands tildes in its configuration file

This scenario verifies that the client expands tildes in pathnames in its configuration file.

given an installed obnam
and file **tilde.yaml**
when I run **obnam --config tilde.yaml config**
then stdout contains home directory followed by **/important**
and stdout contains home directory followed by **/obnam.log**

File: **tilde.yaml**

```
roots: [~/important]
log: ~/obnam.log
server_url: https://backup.example.com
verify_tls_cert: true
```

9.3 Client requires https

This scenario verifies that the client rejects a configuration with a server URL using `http:` instead of `https:`.

given an installed obnam
and file **http.yaml**
when I try to run **obnam --config http.yaml config**
then command fails
and stderr contains "**https:**"

File: **http.yaml**

```
roots: [live]
server_url: http://backup.example.com
verify_tls_cert: true
```

9.4 Client lists the backup schema versions it supports

given an installed obnam
and file **config.yaml**
when I run **obnam --config config.yaml list-backup-versions**
then stdout is exactly "**0.0\n1.0\n**"

9.5 Client lists the default backup schema version

given an installed obnam
and file **config.yaml**
when I run **obnam --config config.yaml list-backup-versions --default-only**

then stdout is exactly "**0.0\n**"

9.6 Client refuses a self-signed certificate

This scenario verifies that the client refuses to connect to a server if the server's TLS certificate is self-signed. The test server set up by the scenario uses self-signed certificates.

given a working Obnam system
and a client config based on **ca-required.yaml**
and a file **live/data.dat** containing some random data
when I try to run **obnam backup**
then command fails
and stderr contains "**self signed certificate**"

File: **ca-required.yaml**

```
verify_tls_cert: true
roots: [live]
```

9.7 Encrypt and decrypt chunk locally

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **cleartext.dat** containing some random data
when I run **obnam encrypt-chunk cleartext.dat encrypted.dat '{"label":"fake"}'**
and I run **obnam decrypt-chunk encrypted.dat decrypted.dat '{"label":"fake"}'**
then files **cleartext.dat** and **encrypted.dat** are different
and files **cleartext.dat** and **decrypted.dat** are identical

9.8 Split a file into chunks

The `obnam chunkify` command reads one or more files and splits them into chunks, and writes to the standard output a JSON file describing each chunk. This scenario verifies that the command works at least in a simple case.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **data.dat** containing "**hello, world**"
and file **chunks.json**
when I run **obnam chunkify data.dat**
then stdout, as JSON, exactly matches file **chunks.json**

File: **chunks.json**

```
[
  {
    "filename": "data.dat",
    "offset": 0,
    "len": 12,
    "checksum": "09ca7e4eaa6e8ae9c7d261167129184883644d07dfba7cbfbc4c8a2e08360d5b"
  }
]
```

Chapter 10

Acceptance criteria for Obnam as a whole

The scenarios in this chapter apply to Obnam as a whole: the client and server working together.

10.1 Smoke test for backup and restore

This scenario verifies that a small amount of data in simple files in one directory can be backed up and restored, and the restored files and their metadata are identical to the original. This is the simplest possible useful use case for a backup system.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I run **obnam list**
then generation list contains <**GEN**>
when I run **obnam resolve latest**
then generation list contains <**GEN**>
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live** restored in **rest** in **rest.yaml**
then manifests **live.yaml** and **rest.yaml** match

File: **smoke.yaml**

```
1 verify_tls_cert: false
2 roots: [live]
```

10.2 Inspect a backup

Once a backup is made, the user needs to be able inspect it to see the schema version.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
and I run **obnam inspect latest**
then stdout contains "**schema_version: 0.0\n**"
when I run **obnam backup --backup-version=0**
and I run **obnam inspect latest**
then stdout contains "**schema_version: 0.0\n**"
when I run **obnam backup --backup-version=1**
and I run **obnam inspect latest**
then stdout contains "**schema_version: 1.0\n**"

10.3 Backup root must exist

This scenario verifies that Obnam correctly reports an error if a backup root directory doesn't exist.

given a working Obnam system
and a client config based on **missingroot.yaml**
and a file **live/data.dat** containing some random data
when I try to run **obnam backup**
then command fails
and stderr contains "**does-not-exist**"

File: **missingroot.yaml**

```
verify_tls_cert: false
roots: [live, does-not-exist]
```

10.4 Back up regular file

The scenarios in this section back up a single regular file each, and verify that is metadata is restored correctly. There is a separate scenario for each bit of metadata so that it's clear what fails, if anything.

All these scenarios use the following configuration file.

File: **metadata.yaml**

```
verify_tls_cert: false
roots: [live]
```

10.4.1 Modification time

This scenario verifies that the modification time is restored correctly.

given a working Obnam system
and a client config based on **metadata.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live** restored in **rest** in **rest.yaml**
then manifests **live.yaml** and **rest.yaml** match

10.4.2 Mode bits

This scenario verifies that the mode ("permission") bits are restored correctly.

given a working Obnam system
and a client config based on **metadata.yaml**
and a file **live/data.dat** containing some random data
and file **live/data.dat** has mode **464**
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live** restored in **rest** in **rest.yaml**
then manifests **live.yaml** and **rest.yaml** match

10.4.3 Symbolic links

This scenario verifies that symbolic links are restored correctly.

given a working Obnam system
and a client config based on **metadata.yaml**
and a file **live/data.dat** containing some random data
and symbolink link **live/link** that points at **data.dat**
and symbolink link **live/broken** that points at **does-not-exist**
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live** restored in **rest** in **rest.yaml**
then manifests **live.yaml** and **rest.yaml** match

10.5 Set chunk size

This scenario verifies that the user can set the chunk size in the configuration file. The chunk size only affects the chunks of live data.

The backup uses a chunk size of one byte, and backs up a file with three bytes. This results in three chunks for the file data, plus one for the generation SQLite file (not split into chunks of one byte), plus a chunk for the generation itself. Additionally, the "trust root" chunk exists. A total of six chunks.

given a working Obnam system
and a client config based on **tiny-chunk-size.yaml**
and a file **live/data.dat** containing "**abc**"
when I run **obnam backup**
then server has **6** chunks

File: **tiny-chunk-size.yaml**

```
verify_tls_cert: false
roots: [live]
chunk_size: 1
```

10.6 Backup or not for the right reason

The decision of whether to back up a file or keep the version in the previous backup is crucial. This scenario verifies that Obnam makes the right decisions.

10.6.1 First backup backs up all files because they're new

This scenario verifies that in the first backup all files are backed up because they were new.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
and I run **obnam list-files**
then file **live/data.dat** was backed up because it was new

10.6.2 All files in second backup are unchanged

This scenario verifies that if a file hasn't been changed, it's not backed up.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**

when I run **obnam backup**
and I run **obnam backup**
and I run **obnam list-files**
then file **live/data.dat** was not backed up because it was unchanged

10.6.3 Second backup back up changed file

This scenario verifies that if a file has indeed been changed, it's backed up.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
given a file **live/data.dat** containing some random data
when I run **obnam backup**
and I run **obnam list-files**
then file **live/data.dat** was backed up because it was changed

10.7 Checksum verification

Each chunk has metadata with the checksum of the chunk contents. This scenario verifies that the client checks the contents hasn't been modified.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam get-chunk <**GEN**>
then exit code is **0**
when chunk <**GEN**> on chunk server is replaced by an empty file
and I invoke obnam get-chunk <**GEN**>
then command fails

10.8 Irregular files

This scenario verifies that Obnam backs up and restores files that aren't regular files, directories, or symbolic links. Specifically, Unix domain sockets and named pipes (FIFOs). However, block and character device nodes are not tested, as that would require running the test suite with `root` permissions and that would be awkward.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data

and a Unix socket **live/socket**
and a named pipe **live/pipe**
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
and I run **obnam restore latest rest**
given a manifest of the directory **live** restored in **rest** in **rest.yaml**
then manifests **live.yaml** and **rest.yaml** match

10.9 Tricky filenames

Obnam needs to handle all filenames the underlying operating and file system can handle. This scenario verifies it can handle a filename that consists on a single byte with its top bit set. This is not ASCII, and it's not UTF-8.

given a working Obnam system
and a client config based on **metadata.yaml**
and a file in **live** with a non-UTF8 filename
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live** restored in **rest** in **rest.yaml**
then manifests **live.yaml** and **rest.yaml** match

10.10 Unreadable file

This scenario verifies that Obnam will back up all files of live data, even if one of them is unreadable. By inference, we assume this means other errors on individual files also won't end the backup prematurely.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
and a file **live/bad.dat** containing some random data
and file **live/bad.dat** has mode **000**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
then file **live/data.dat** is restored to **rest**
and file **live/bad.dat** is not restored to **rest**

10.11 Unreadable directory

This scenario verifies that Obnam will skip a file in a directory it can't read. Obnam should warn about that, but not give an error.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/unreadable/data.dat** containing some random data
and file **live/unreadable** has mode **000**
when I run **obnam backup**
then stdout contains "**live/unreadable**"
and backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
then file **live/unreadable** is restored to **rest**
and file **live/unreadable/data.dat** is not restored to **rest**

10.12 Unexecutable directory

This scenario verifies that Obnam will skip a file in a directory it can't read. Obnam should warn about that, but not give an error.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/dir/data.dat** containing some random data
and file **live/dir** has mode **600**
when I run **obnam backup**
then stdout contains "**live/dir**"
and backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
then file **live/dir** is restored to **rest**
and file **live/dir/data.dat** is not restored to **rest**

10.13 Restore latest generation

This scenario verifies that the latest backup generation can be specified with literal string "latest". It makes two backups, which are different.

given a working Obnam system
and a client config based on **metadata.yaml**
and a file **live/data.dat** containing some random data
when I run **obnam backup**
given a file **live/more.dat** containing some random data
and a manifest of the directory **live** in **second.yaml**
when I run **obnam backup**
and I run **obnam restore latest rest**
given a manifest of the directory **live** restored in **rest** in **rest.yaml**
then manifests **second.yaml** and **rest.yaml** match

10.14 Restore backups made with each backup version

given a working Obnam system
and a client config based on **metadata.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I run **obnam backup --backup-version=0**
and I run **obnam restore latest rest0**
given a manifest of the directory **live** restored in **rest0** in **rest0.yaml**
then manifests **live.yaml** and **rest0.yaml** match
when I run **obnam backup --backup-version=1**
and I run **obnam restore latest rest1**
given a manifest of the directory **live** restored in **rest1** in **rest1.yaml**
then manifests **live.yaml** and **rest1.yaml** match

10.15 Back up multiple directories

This scenario verifies that Obnam can back up more than one directory at a time.

given a working Obnam system
and a client config based on **roots.yaml**
and a file **live/one/data.dat** containing some random data
and a file **live/two/data.dat** containing some random data
and a manifest of the directory **live/one** in **one.yaml**
and a manifest of the directory **live/two** in **two.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live/one** restored in **rest** in **rest-one.yaml**
and a manifest of the directory **live/two** restored in **rest** in **rest-two.yaml**
then manifests **one.yaml** and **rest-one.yaml** match
and manifests **two.yaml** and **rest-two.yaml** match

File: **roots.yaml**

```
roots:
- live/one
- live/two
```

10.16 CACHEDIR.TAG support

10.16.1 By default, skip directories containing CACHEDIR.TAG

This scenario verifies that Obnam client skips the contents of directories that contain CACHEDIR.TAG[1], but backs up the tag itself. We back up the tag so that after a restore, the directory continues to be tagged as a cache directory.

given a working Obnam system
and a client config based on **client.yaml**
and a file **live/ignored/data.dat** containing some random data
and a cache directory tag in **live/ignored**
and a file **live/not_ignored/data.dat** containing some random data
and a manifest of the directory **live/not_ignored** in **initial.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live/not_ignored** restored in **rest** in **restored.yaml**
then manifests **initial.yaml** and **restored.yaml** match
and file **rest/live/ignored/CACHEDIR.TAG** contains "**Signature: 8a477f597d28d172789f06886806bc55**"
and file **rest/live/ignored/data.dat** does not exist

File: **client.yaml**

```
roots:
- live
```

10.16.2 Incremental backup errors if it finds new CACHEDIR.TAGs

To mitigate the risk described in the "Threat Model" chapter, Obnam should notify the user when it finds CACHEDIR.TAG files that aren't present in the previous backup. Notification is twofold: the path to the tag should be shown, and the client should exit with a non-zero code. This scenario runs backups the a directory (which shouldn't error), then adds a new tag and backups the directory again, expecting an error.

given a working Obnam system
and a client config based on **client.yaml**
and a file **live/data1.dat** containing some random data
and a file **live/data2.dat** containing some random data
when I run **obnam backup**
then exit code is **0**
given a cache directory tag in **live/**
when I try to run **obnam backup**

[1] `https://bford.info/cachedir/`

then exit code is **1**
and stdout contains "**live/CACHEDIR.TAG**"
when I run **obnam list-files**
then exit code is **0**

then file live/CACHEDIR.TAG was backed up because it was new and stdout doesn't contain "live/data1.dat" and stdout doesn't contain "live/data2.dat"

10.16.3 Ignore CACHEDIR.TAGs if `exclude_cache_tag_directories` is disabled

This scenario verifies that when `exclude_cache_tag_directories` setting is disabled, Obnam client backs up directories even if they contain CACHEDIR.TAG[2]. It also verifies that incremental backups don't fail when new tags are added, i.e. the aforementioned mitigation is disabled too.

given a working Obnam system
and a client config based on **client_includes_cachedirs.yaml**
and a file **live/ignored/data.dat** containing some random data
and a cache directory tag in **live/ignored**
and a file **live/not_ignored/data.dat** containing some random data
and a manifest of the directory **live** in **initial.yaml**
when I run **obnam backup**
then backup generation is **GEN**
when I invoke obnam restore <**GEN**> **rest**
given a manifest of the directory **live** restored in **rest** in **restored.yaml**
then manifests **initial.yaml** and **restored.yaml** match
given a cache directory tag in **live/not_ignored**
when I run **obnam backup**
then exit code is **0**
and stdout doesn't contain "**live/not_ignored/CACHEDIR.TAG**"

File: **client_includes_cachedirs.yaml**

```
roots:
- live
exclude_cache_tag_directories: false
```

10.17 Generation information

This scenario verifies that the Obnam client can show metadata about a backup generation.

given a working Obnam system
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data

[2] `https://bford.info/cachedir/`

and a manifest of the directory **live** in **live.yaml**
and file **geninfo.json**
when I run **obnam backup**
and I run **obnam gen-info latest**
then stdout, as JSON, has all the values in file **geninfo.json**

File: **geninfo.json**

```
{
    "schema_version": {
        "major": 0,
        "minor": 0
    },
    "extras": {
        "checksum_kind": "sha256"
    }
}
```

Chapter 11

Acceptance criteria for backup encryption

This chapter outlines scenarios, to be implemented later, for verifying that Obnam properly encrypts the backups. These scenarios verify only encryption aspects of Obnam.

11.1 Backup without passphrase fails

Verify that trying to backup without having set a passphrase fails with an error message that clearly identifies the lack of a passphrase.

given a working Obnam system
and a client config, without passphrase, based on **encryption.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I try to run **obnam backup**
then command fails
and stderr contains "**obnam init**"

File: **encryption.yaml**

```
verify_tls_cert: false
roots: [live]
```

11.2 A passphrase can be set

Set a passphrase. Verify that it's stored in a file that is only readable by it owner. Verify that a backup can be made.

given a working Obnam system
and a client config, without passphrase, based on **encryption.yaml**
and a file **live/data.dat** containing some random data
and a manifest of the directory **live** in **live.yaml**
when I run **obnam init --insecure-passphrase=hunter2**
then file **.config/obnam/passwords.yaml** exists
and file **.config/obnam/passwords.yaml** is only readable by owner
and file **.config/obnam/passwords.yaml** does not contain "**hunter2**"

11.3 A passphrase stored insecurely is rejected

Verify that a backup fails if the file where the passphrase is stored is readable by anyone but its owner. Verify that the error message explains that the backup failed due to the passphrase file insecurity.

11.4 The passphrase can be changed

Verify that the passphrase can be changed and that backups made before the change can no longer be restored. (Later, this requirement will be re-evaluated, but this is simple and gets us started.)

11.5 The passphrase is not on server in cleartext

Verify that after the passphrase has been set, and a backup has been made, the passphrase is not stored in cleartext on the server.

11.6 A backup is encrypted

Verify that the backup repository does not contain the backed up data in cleartext.