Obnam2—a backup system

Lars Wirzenius

2020-11-27 09:11

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.

Contents

Chapter 1

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].

1.1 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 2

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.
- *Not done:* **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.
- *Not done:* **Detects corruption:** if a file in the repository is modified or deleted, the software notices it automatically.
- *Not done:* **Repository is encrypted:** all data stored in the repository is encrypted with a key known only to the client.
- *Not done:* **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.

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 3

Software architecture

3.1 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:**
- **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.

3.2 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.

3.3 On "btrfs send" and similar constructs

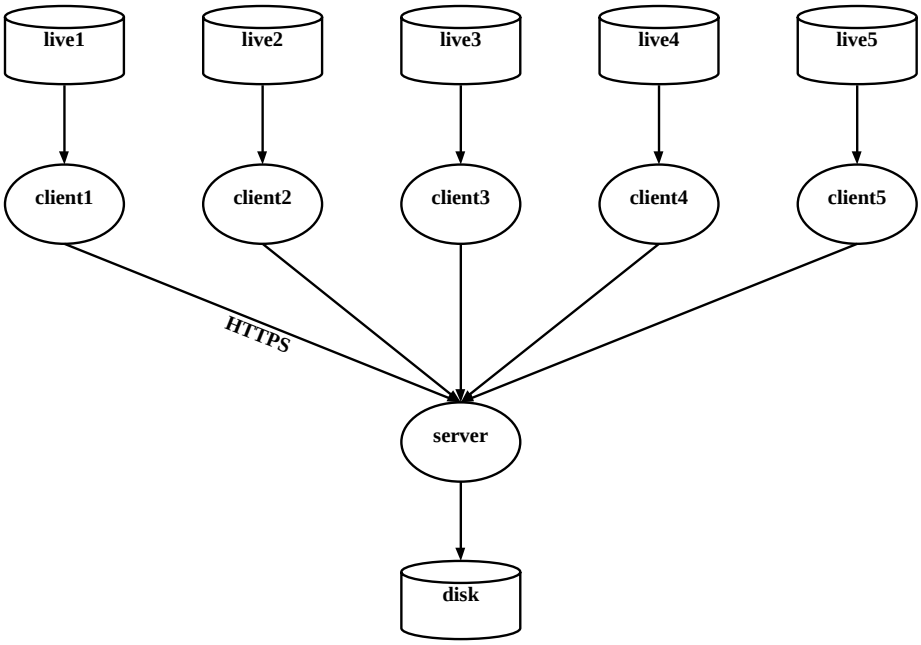
The btrfs and ZFS file systems, and possibly others, have a way to mark specific states of the file system and efficiently generate a "delta file" of all the changes between the states. The delta can be transferred elsewhere, and applied to a copy of the file system. This can be quite efficient, but 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.

3.4 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 file, which represents a backup generation, 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.

Chapter 4

Implementation

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

4.1 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 fields:

- `sha256` – 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
- `generation` – set to `true` if the chunk represents a generation
  - may also be set to `false` or `null` or be missing entirely
  - the server allows for listing chunks where this field is set to `true`
- `ended` – the timestamp of when the backup generation ended
  - note that the server doesn't process this in any way, the contents is entirely up to the client
  - may be set to the empty string, `null`, or be missing entirely
  - this can't be used in searches

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.

4.2 Server

The server has the following API for managing chunks:

- `POST /chunks` – store a new chunk (and its metadata) on the server, return its randomly chosen identifier
- `GET /chunks/<ID>` – retrieve a chunk (and its metadata) from the server, given a chunk identifier
- `GET /chunks?sha256=xyzzy` – find chunks on the server whose metadata indicates their contents has a given SHA256 checksum
- `GET /chunks?generation=true` – find generation chunks

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.

When searching for chunks, any matching chunk's identifiers and metadata are returned in a JSON object:

```
{
  "fe20734b-edb3-432f-83c3-d35fe15969dd": {
     "sha256": "09ca7e4eaa6e8ae9c7d261167129184883644d07dfba7cbfbc4c8a2e08360d5b",
     "generation": null,
     "ended: null,
  }
}
```

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

4.3 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

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

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 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.

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

Chapter 5

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.

5.1 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 an installed obnam
and a running chunk server
and a file **data.dat** containing some random data
when I POST **data.dat** to **/chunks**, with **chunk-meta**: **{"sha256":"abc"}**
then HTTP status code is **201**
and **content-type** is **application/json**
and the JSON body has a field **chunk_id**, henceforth **ID**

We must be able to retrieve it.

when I GET /chunks/<**ID**>
then HTTP status code is **200**
and **content-type** is **application/octet-stream**
and **chunk-meta** is **{"sha256":"abc","generation":null,"ended":null}**
and the body matches file **data.dat**

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

when I GET /chunks?sha256=**abc**
then HTTP status code is **200**

and **content-type** is **application/json**
and the JSON body matches **{"<ID>":{"sha256":"abc","generation":null,"ended":null}}**

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 /chunks/<**ID**>
then HTTP status code is **200**
when I GET /chunks/<**ID**>
then HTTP status code is **404**
when I GET /chunks?sha256=**abc**
then HTTP status code is **200**
and **content-type** is **application/json**
and the JSON body matches **{}**

5.2 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 an installed obnam
and a running chunk server
when I try to GET /chunks/**any.random.string**
then HTTP status code is **404**

5.3 Search without matches

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

given an installed obnam
and a running chunk server
when I GET /chunks?sha256=**abc**
then HTTP status code is **200**
and **content-type** is **application/json**
and the JSON body matches **{}**

5.4 Delete chunk that does not exist

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

given an installed obnam
and a running chunk server
when I try to DELETE /chunks/**any.random.string**
then HTTP status code is **404**

Chapter 6

Smoke test for Obnam as a whole

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 an installed obnam
and a running chunk server
and a client config based on **smoke.yaml**
and a file **live/data.dat** containing some random data
when I run **obnam backup smoke.yaml**
then backup generation is **GEN**
when I run **obnam list smoke.yaml**
then generation list contains <**GEN**>
when I invoke obnam restore **smoke.yaml** <**GEN**> **rest**
then data in **live** and **rest** match

File: **smoke.yaml**

```
1 root: live
```