A block-based storage model for remote online backups in a trust-no-one environment

http://www.duplicati.com/
Kenneth Skovhede (author, kenneth@duplicati.com)
René Stach (editor, rene@duplicati.com)

February 2014

Abstract

In Duplicati 2.0 we introduce a new block storage format to store the users' backups on remote file servers. This document describes the drawbacks of the old storage format used in Duplicati 1.3 and how these are addressed in a new block storage format for Duplicati 2.0. The document provides detailed explanations how the new block storage format stores backup data and how files can be restored from it. Finally, the document discusses quickly a few major ideas about backup software functionality and their compatibility with the new storage format.

Motivation

The Duplicati storage system is based on the efficient rdiff algorithm, which enables very compact change detection, especially for files where the content moves, such as text files. By using the rdiff algorithm, Duplicati is capable of calculating byte-by-byte differences in files, and storing these as patches. Such a patch is able to turn an older version of a file into a newer version. This approach is very space efficient, both in terms of used local storage but also in terms of used remote storage.

Unfortunately, this space efficient approach has some drawbacks. When a file frequently changes, there is a huge number of patches. As Duplicati always compares the current version of a file to the latest version of the same file in the backup any patch depends on the previous patch, which again depends on the previous patch. This continues until a patch eventually depends on the first version, a full file, in the backup. A number of dependant files and patches is called a chain. The entire approach is called incremental backup.

Unfortunately, there is an unsolved problem with performing backups in this way. Over time the difference between the original file and the latest version becomes very large as the file evolves. This means that either the size of patches grows or the chain of patches gets very long. These chains have the problem that they are slow to process, and since the chain links depend on each other, a single broken link will make the rest of the chain useless. As all patches depend on the initial file, nothing must be deleted and the backup chain grows and grows and grows. At the same time the risk grows that due to a malfunction of the storage the chain breaks and the file gets lost.

Another challenge is to make sure that Duplicati works with many different storage providers. Therefore, it uses an abstraction to storage, that allows only four commands to be issued: PUT, GET, LIST and DELETE. This abstraction ensures that virtually any file based backend can be used, but it also imposes some restrictions on the allowed solutions. For instance, the rsync system has a server-side component, which enables it to rewrite the changed parts of a file. As Duplicati does not have such a server component, it can only upload the entire file, even if only a small portion of the file has changed.

Since Duplicati is based on this concept of a “dumb” storage backend, Duplicati has to cope with some restrictions. For instance, it is not possible to modify existing files which makes it impossible to update an old existing backup. To update old data, Duplicati has to upload new data and delete the old data. This is solved by grouping data into volumes that can then be encrypted and stored. While this works, it also sets some limitations on what solutions are possible.

Even though Duplicati uses a very standard way of creating full and incremental backups, there are some severe drawbacks:

- On the one hand, each full backup is a real pain for any serious dataset. On the other hand, increasing the time between the full backups introduces longer restore chains and the risk of failure. Users have to find a good trade-off between these two issues themselves (which still does not solve the problem).
- The entire concept makes it impossible to discard specific old data. When a backup grows over time, users usually want to delete the oldest backups first which is not possible with the chain of dependant patches. Furthermore, it is likely that users want to restore the newest version of the file, which currently requires to process the entire chain.

Modern file formats tend to store compressed data. E.g. Libre Office stores documents in zip files. So does Microsoft Office. Graphics are compressed in JPEG files and movies compressed using H.264 or other compression techniques. The problem with compression is, it increases data entropy, i.e. the output of the compression is similar to random data. A single changed byte of data in compressed files can cause the entire file to change. When using the rdiff algorithm there is very little chance of finding similar chunks of data in compressed data.

Many backup sets today are dominated quite heavily by multimedia content, such as photos, music, and video. Most multimedia files change very rarely, if ever, but when they change they are usually compressed and thus rdiff has very little chance of finding similarities. This means that the files that comprise the bulk of the backup data are either repeatedly uploaded as unchanged, or is uploaded as a patch that is around the same size as the original.

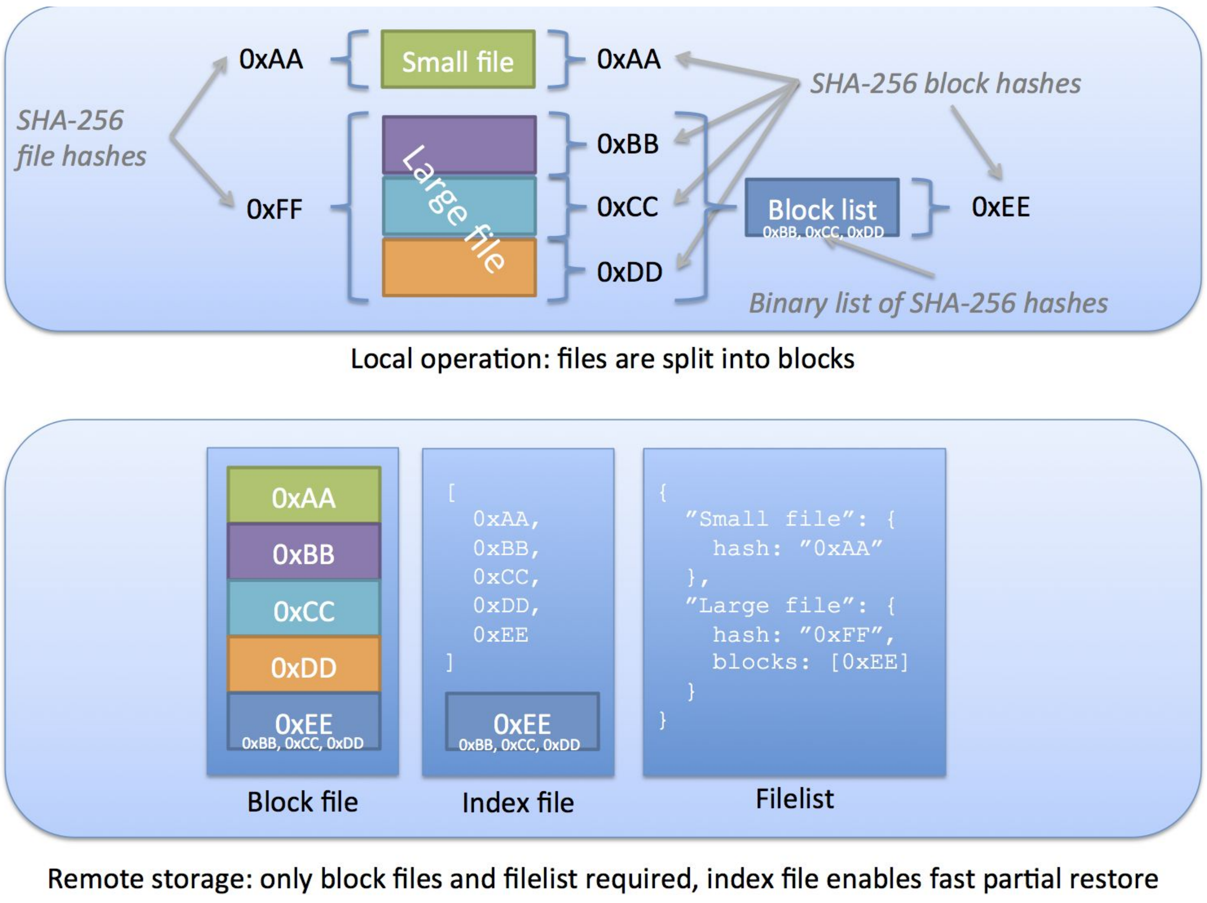
Another issue in the current solution is that the volume names are used to establish chain relationship, which turned out to be problematic for some storage solutions, where the listing of files is unreliable. While this can be improved with various workarounds, the chains inherently need a structure, which would eventually require storing this information in files instead of encoding it in the filenames.

Solution

After numerous attempts and suggestions to improve the current system, the decision was made to go for revolution instead of evolution and develop a block based storage format that addresses many of the issues the old storage had and that would naturally cause identical blocks to only be stored once. This is called deduplication and also provides features like "detect moved or renamed files". The new block storage is in some way inspired by the way bigger online services like DropBox work, but without requiring the server-side database.

The basic idea is to split each file into blocks of a fixed size. The final block may be smaller than the others, if the filesize is not evenly divisible with the block size. The hash of each block is then recorded, and each file can be described by a list of hashes. This format naturally provides deduplication, as there is no need to store blocks with the same hash twice. We chose the SHA-256 hash as that seems to be a safe bet, although the cryptographic properties of SHA are not required. The hashing algorithm can be selected before running the initial backup, and Duplicati currently also supports the MD5 and SHA-512 hashing algorithms.

To reconstruct files, the client needs to fetch the list of hashes for that particular file, and then retrieve the needed blocks. To avoid downloading large volumes, it makes sense to decouple the file description from the actual data. This leads to two different file types, blocks and filelists, as shown in figure 1.



```
[
  0xAA,
  0xBB,
  0xCC,
  0xDD,
  0xEE
]
```

```
{
  "Small file": {
    hash: "0xAA"
  },
  "Large file": {
    hash: "0xFF",
    blocks: [0xEE]
  }
}
```

Figure 1 - Graphical overview of blocks and file types in Duplicati

However, there are two complications relating to this setup. One problem is that the list of hashes can become very large. For a setup with a block size of 100KiB using an SHA-256 hash, a 1TiB file will fill up more than 300MiB hash data, which would need to be stored in the filelist. That was the reason, to store the hash lists as binary blobs. As each SHA-256 hash is 32 bytes, a list of hashes is simply the 32 bytes written, then the next 32 bytes, etc. The hash lists are stored as blocks in the same way that the actual data is stored. This reduces the size of the hash list with a factor of blocksize / hashsize, or for the same 1TiB, the hash list in the filelist becomes around 100KiB. This introduces a level of indirection, but does not add further file or data types. If a single byte is changed in the 1 TiB file, the block itself needs to be updated, as well as the binary blob that contains the block's hash, resulting in two blocks being updated.

The other complication is, that over time, the volumes that contain the blocks will start to contain unused data. That is why an approach was taken to periodically compact volumes. Compacting replaces volumes with large amount of unused data with new volumes that only contain used data. This compacting can take place regularly which adds some bandwidth usage to the backup process but frees storage space. The actual algorithm can change over time without affecting the file format.

The initial version is to calculate the size of blocks that cannot be reached in each volume, and once a certain threshold is exceeded, the volume is marked as wasteful. As soon as the overall waste exceeds the size of a volume, wasteful volume(s) are downloaded and new volumes are written. As the waste exceeded the size of one volume, the number of volumes is automatically decreased. In some cases the non-wasted blocks can be found in local files, which makes it possible to create the volume without downloading data. It is possible to do some strategic compacting that will exploit the fact that old blocks are less likely to become obsolete than new blocks.

Reconstruction

In the implementation all information is stored in a local SQLite database, with the exception of the actual data blocks. This enables all queries to be performed locally with SQL commands. The database is essentially a replica of the remote storage, meaning that it is possible to build the remote storage from the database and vice versa (excluding the actual data blocks).

When reconstructing files (restoring data), the database is queried to obtain all required blocks. Initially the destination files are scanned (if they exist) for matching blocks. This enables "updating" or "reverting" existing folders by only touching required blocks. Then the database is queried to find potential local files that may hold the desired blocks. Should such blocks be found, they are used to patch the target files. Finally the remote volumes are downloaded as required to provide the missing blocks.

In the case that the local database does not exists, i.e. when restoring from a new machine, the database is rebuilt before running the restore operation. To avoid downloading all the block volumes, a third file type is added: index files. Index files simply contain a list of blocks and their sizes in the block volumes. This information can be used to establish block/volume mappings. Before the files can be restored, the hash lists for the files must be known. As explained these lists can become quite large, so they are stored in the block volumes. It is possible to store these block lists in the index files as well, causing a bigger overhead, but reduces restore time. If the block lists are not found in the index files, the block volumes are downloaded and the lists extracted. To avoid downloading files twice, a partial restore operation is executed after each download. This will catch many cases where the blocklists and the data is located in the same volume, but there may be cases where this is not true. It is possible to employ a download cache to further limit this issue.

If a required block is not found after scanning the index files, block files are downloaded at random, until all known blocks are found. This enables a restore, even in cases without any index files.

Since this format does not employ chains, there is no direct link between any files. This makes it possible to work even with faulty services, and makes partial recovery much simpler. The filenames are no longer encoded with any information, except the file type. The name of the file list also includes an approximate time of creation, because this makes it possible to select the desired file list when no local database exists, without downloading all file lists.

Extra features

There are some ideas around Duplicati that come up again every now and then. Besides the issues that are addressed already using the new storage system there are some more worth to be commented here:

Extended metadata. In Duplicati, the only metadata that is stored is the file modification time. In the new file format, metadata is a data type of its own, and is implemented in an extensible manner. In the basic version, only the file attributes and the write timestamp is stored. It is very simple to plug-in new metadata handlers so things like ACL and others can be stored. The Alternate Data Streams and OSX resource forks are also considered in the implementation but not fully implemented.

Continuous backup. Since the overhead of any backup is essentially the filelist, it becomes almost feasible to run continuous backups. This can be made with even less overhead, by implementing some form of “update” mechanism that distributes the filelist into “base” and “patch”. Although this is not unlike the original full/incremental approach, the datasets are much smaller and the overhead of a full filelist is much more manageable. Keeping a cache of small volumes, also makes it possible to reduce the cost of compacting data frequently.

Shared deduplication. By using index files, it becomes possible for multiple backups to utilize the same block volumes. This can enable deduplication across multiple backups, potentially from distributed locations.

Cloud storage. Even though Duplicati is not intended to be used as a general purpose cloud storage system, the continuous backup and the shared deduplication are two strong components for making a DropBox like client. There needs to be some conflict detection/handling and more efficient change propagation, but it is otherwise possible.

Data loss prevention. The entire system is designed to allow random loss/corruption of pieces and still perform a best-effort restore. That is, even though some blocks are missing or some part of the filelist is bad, a partial restore is possible. But with the deduplication each block can suddenly be important for multiple files. To prevent this, a parity system is proposed that will upload parity files that can help recreate broken files.