Design and Implementation of Website Backup as a Service

Ricardo Persoon

July 22, 2016

Supervisors:
dr. Ir. F. Verbeek
dr. K. F. D. Rietveld

Abstract

Maintaining reliable backups of a website can be time consuming, expensive, or both. Numerous approaches exist for website backup, each with their specific limitations. In this thesis, we propose and design a new sophisticated website backup application, which combines the best features of existing solutions and satisfies a number of requirements. We will implement a working prototype, describe its operation and explain the structure and design decisions.

Contents

1 Introduction

Creating backups remains a challenge for many website owners. Not particularly due to technical or theoretical complexity, but mainly because it is time consuming to set up backups, while it is not beneficial or profitable until disaster strikes. Typical practical limitations are the additional costs of storage space, bandwidth, and the continuously returning effort of verifying the proper functioning of backup processes.

Having backups readily available can be useful in many situations. Website often get hacked, not only by professional hackers but by bot networks as well, which target commonly used content management systems (CMS) using public known flaws in outdated versions. Furthermore, hardware of a hosting provider can fail, a website update could go wrong or code might be accidentally deleted during programming or maintanance.

Hosting providers often offer backups themselves, even included with standard hosting products. However, those backups are often being stored in the same datacenter or even on the server systems where the website itself is located, and will thereby not protect against disasters on a datacenter level. Furthermore, both the website and backups will be lost if the hosting provider goes bankrupt. Research also shows that 43% of website losses are blamed on the hosting company [1], thus relying on their backups might not be ideal. Having a reliable independent backup solution is critical for the continued stability of a website.

For our project, a website backup is considered to be a copy of the files and databases belonging to the website. The server-side files and software are not included. A considerable amount of approaches and solutions are available to create such website backups. We will review some of the available options and discuss their pros and cons.

The most straightforward backup approach is to periodically copy all data to another computer or external storage device by hand, continuously requiring manual action. The copying could be automated using a small script or in some cases a plugin, often offered by content management systems like WordPress or Drupal. However, copying all data over and over again is not economical at all, as it consumes large quantities of bandwidth and storage space, which will be either expensive or limit the number of historical copies that can be preserved.

Often only small incremental changes in files or database data occur over time, which can be exploited to improve backup processes and storage consumption. Data traffic could be significantly reduced using rsync or similar tools,

which only transfer changes. As to storage, many use a revision control system to store the increasing amount of data efficiently. Although originally intended to keep track of source code while programming, software like Git can drastically reduce storage consumption while preserving many revisions of your data. One could create a repository, periodically copy the web and database data and commit after each backup.

Unfortunately, revision control systems limit your ability to manage the stored data. In particular, it is not possible to delete or merge intermediate backups by design, which is inconvenient if you pursue a more sophisticated backup strategy. It will not be possible to deploy a backup storage schedule which for example maintains daily backups for a month, weekly backups for a year and monthly backups thereafter. Pruning the first backup proves to be challenging as well, leading to ever increasing storage consumption.

Finally, commercial website backup applications exist, which solve many of aforementioned problems. The current best known website backup service is Codeguard [2], a United States based corporation specialized in website and database backup. Backups are continuously created and a notification is sent in case of any irregularities, which relieves people of most of the manual efforts. Codeguard internally uses Git to store backup data, which they altered to be able to delete backups at the tail. Deleting intermediate backups is not possible, limiting the available backup retention strategies. Another drawback of all available commercial website-backup applications is the security of your data. They do not explicitly encrypt your data in a way such that only the end user is able to decrypt his own data, which is regrettable in the case of a data breach or data requests from government agencies.

An overview of the discussed backup solutions and their features has been included in table 1.

	Incremental storage	Encrypted by default	Easily scalable	Deleting intermediate backups	Not accessible by others
Manual copy				v	v
Automated copy				v	v
CMS plugins				v	v
Git	v				v
Commercial applications	v	v	v		

Table 1: The features available with each backup approach.

In this study, we will propose and design a *software-as-a-service* backup application that takes care of most obstacles. The application is required to be reliable, fault-tolerant and secure. In particular, we specify the following requirements:

- Only involve users in backup activities during the backup setup and in the case of problems. Periodical backups must be performed automatically as scheduled without any manual action. In the case of complications, which can include failed backups or changed login credentials, alerts must be send to notify the user.
- A redundant storage back end which prevents data-loss due to physical and non-physical causes. The storage facilities should be able to withstand data corruption and hard disk or entire server failures. Moreover, there should be no loss of data in the unlikely event of an entire data center location being destroyed due to flooding or fire. Maintaining physical seperated duplicates of all backup data is essential.
- A level of scalability that supports thousands of websites and backups without any major changes to the application itself. Therefore, storage facilities should be easily expandable
- Data encryption in a way such that users have the possibility to exclude

everyone, including the administrators of the application, to access their data while it is stored in the application's storage facilities.

We will concentrate on creating backups for websites and the most commonly used database systems. In particular, we will consider all small and medium-sized websites with website files located on one server and possibly one or multiple related databases. However, the resulting application can be used to backup any other type of data, as long as it can be remotely accessed using any of the supported connection methods.

The thesis is organized as follows. In the first chapter we will explain the functionality of several algorithms and design choices belonging to backup creation, which includes the file and database backup algorithms and the encryption. Thereafter, we will elaborate on the infrastructure and overall operation of the application.

2 Efficient backups, storage and encryption

Scalability and reliability are fundamental requirements of this project. Efficient backup algorithms are desired to support processing of many websites at once, while limiting the required resources to a minimum and maintaining the integrity of every single backup.

We have designed several techniques in order to meet these requirements. In this chapter, we will discuss the implementation of the backup and encryption mechanisms. We will distinguish between file and database backup, as entirely different approaches are required due to their data structures and connection methods.

2.1 File backup

Our algorithm for creating file backups can be split into two steps. A data structure is produced first, which is used as an index of the remote website files. Using this structure, we can efficiently determine which files have been added or changed. The second phase follows, in which new files are transferred to long term backup storage and changes are administered in a database.

2.1.1 Indexing available files

Every backup starts by indexing the files on the remote server. A webserver can usually be approached using several protocols, of which FTP and a secured SSH file transfer connection (SFTP) are most common. A tree-like structure including all directories available for backup is produced, in which all files and subdirectories in a directory are determined. Additionally, for each file the name, size, permissions and timestamp of last modification are collected.

By comparing this tree to our database record of the most recent backup, we can determine which files have since been modified or added. A file is considered unchanged if the timestamp of last modification, its file size and the permissions did not change, whereupon a file is not downloaded. Naturally, all files are transferred during the first backup.

2.1.2 File storage

The integrity of backup data or associated metadata is critical. As specified in the requirements, we wish each piece of data to be stored at two or more separate physical locations. Therefore, data replication is an important element of the file storage backend, which limits the available options.

The possibility of using ordinary servers containing a Linux operating system and a file system that supports snapshots was initially investigated. A snapshot is the state of a storage volume at a point in time, therefore allowing us to maintain backup revisions by creating a snapshot after each backup. We tried ZFS [3], a modern file system with snapshot support, excellent data verification features and sophisticated built-in software RAID support, which allows the use of RAID without an expensive hardware-based RAID card.

ZFS turned out to be an excellent tool for data storage and point-in-time snapshots. On the other hand, scalability and replication to multiple servers proved to be difficult. ZFS is a local filesystem and cannot be extended over multiple servers, which causes capacity issues as backups usually grow over time. Migrating ZFS volumes between servers in an effort to balance data is possible but an undesired process which can fail in many ways. As to replication, ZFS supports the transfer of snapshots over different servers, but it often failed during our testing. Hence, ZFS seemed to be unsuitable.

The projects GlusterFS [4] and Ceph [5] were examined as well. Being distributed filesystems, they are able to emulate one single filesystem using multiple servers, which can easily be expanded by adding additional storage hardware. Drawbacks of a distributed filesystem are the highly constrained support for snapshots and difficulties to maintain physical separated copies of data, as network latency is a serious issue with distributed filesystems.

The last alternative, an object store, turned out to suit our needs the best. Designed and used by Facebook to store photos and video's [7], object stores increase in popularity and nowadays different implementations exist. Being of an higher level architecture, it relieves us of implementing replication and balancing ourselves. An object store can be considered as an infinitely scalable black box, accommodating so-called containers: essentially folders containing files or *objects*. Its backend consists of a number of storage servers, their number expandable as desired on the fly. Data transfer to and from the object store proceeds using an application programming interface (API).

Redundancy and physical data separation is fully managed by the object store. Storage servers can be grouped in locations, and one can specify objects to be replicated a number of times in different locations. Objects in a specific container are distributed over the entire storage cluster and not contained on a single machine. This allows containers to grow infinitely and capacity can simply be increased by adding new storage machines to the pool.

2.1.3 File metadata

The objectstore is only capable of storing the actual files. Corresponding metadata and backup revision information must therefore be administered in an alternative way.

At first we considered MongoDB [6] for metadata storage. Being a NoSQL database, MongoDB stores its data as so-called documents instead of rows. It supports automatic sharding - dividing data over multiple servers - and replication sets, thereby facilitating our requirements without much need for manual implementation.

However, a MongoDB trial setup with actual backup data turned out to be slow on our specific queries, even with maximum indexes. The structure of our data did not work out well with the document orientated storage of NoSQL and MongoDB in particular.

More desirable results were obtained with a setup using MySQL servers. Performance was more than sufficient and a separate database could be maintained for the metadata of each website to improve effectiveness even more. Unfortunately, MySQL does not support true sharding, so we had to implement that ourselves.

A *metadata database* is created for each website, which is capable of maintaining all file and directory information. When a new file is found during backup creation, the file is administered in the database with a so called *starting revision*, the number of the current backup. If the file remains unchanged during subsequent backups, nothing has to be changed in the database either. When the file is changed or removed, the *ending revision* is recorded. This way, the size of the metadata database is limited, while maintaining a transparant and fast method of determining the files in a specific backup at all times.

An additional deduplication strategy has been implemented to reduce storage consumption even more. For each file an MD5 checksum will be calculated before transfer to the object store, which is compared to the available checksums of files already in storage. If a file with a matching checksum and size is found, those files must be identical. This can happen if the same file is used in different folders of the website or when a file is removed in an earlier revision and readded later on. Identical files will only be stored once using this deduplication technique.

Due to our encryption scheme, deduplication can only be applied on the scope of single websites. As users are able to encrypt their files with their own password, we are unable to decrypt those files if another user requested a restore. Therefore duplicate files shared between users cannot be deduplicated, and for

the sake of simplicity and portability of websites, deduplication has been limited to single websites. This obviously reduces the deduplication effectivity, but our results - as shown in the Results chapter - show sufficient savings to justify the use of deduplication.

The database schema of a metadata database can be found in Appendix A.

2.2 Database backup

Database backups require an entirely different approach, as the database shell and dump tools have to be used for all backup and restore operations. Incremental backups are usually not possible without access to the database server itself, because they require manual modifications to the binary log or other internal features of the database server. Automatic operation without any adjustments to the software on the remote web or database server is important, therefore modifications to standard configurations are not possible.

These restrictions limit us to generating full dumps of a database or table for every single backup. Such a database dump contains CREATE and INSERT statements, on the basis of which the entire database can be reconstructed.

However, database data of the average website usually does not change much over time. Content is only occasionally added or modified and, in general, the database grows due to user activity or logging. Storing many database dumps containing mostly the same data would therefore be an unnecessary waste of storage space. We can conserve lots of space using a widely available tool: diff. The diff utility is able to calculate the difference between two files. An original file can be reconstructed using diff's counterpart patch, which restores a file when applied to an original file and a patch file generated by diff.

The principle of creating patch files using diff can be applied to our periodic database dumps. A patch file can be manufactured using the previous and new backup, which in turn can be applied to the previous backup in order to reconstruct the current backup dump. Therefore, we only have to keep one full backup, while successive backups can be reconstructed using their corresponding patch files. In case of a database restore, the most recent full backup can be collected, upon which all successive patches are applied until the required database dump has been reassembled. For backup creation, it is convenient to save the most recent dump as full backup, for the reason that it is required to construct a patch file when a new backup is created.

A small adjustment to the database dump is required for the diff and patch method to function properly. By default, a database dump will contain one

(possibly very large) INSERT statement for each table, with all of the table's rows at one line of the dump. This does not work out well for the diff tool, which compares files line by line and will therefore mark an entire table changed if only a single byte is modified. This can be addressed by letting the database dump generate separate INSERT statements for each of the rows, whereby each table row occupies a line and changes are dealt with per table row instead of entire tables. For the MySQLdump tool, this can be realized by setting the extended-insert=FALSE flag. As the dump files are compressed, the additional data consumption of repeated INSERT statements is negligible compared to the savings of the incremental strategy.

Continuously creating incremental database backups could lead to long restore times. In case of a restore of backup number 100, 99 incremental backups would have to be patched if backup 1 is the most recent full backup. Moreover, all following backups up to the next full backup would be lost in case of a lost or corrupt incremental backup. Therefore we maintain a full database dump after a specified number of incremental backups. Determining the optimal number of incremental backups before a new full backup should be preserved is difficult: it is a tradeoff between reduced use of storage space, and faster restore times and less risk of data corruption. We have determined this number to be 25 for now. Figure 1 contains a visualization of the available backups of an arbitrary website after the first 7 backup iterations, with the number of incremental backups before a new full backup is maintained set to 3.

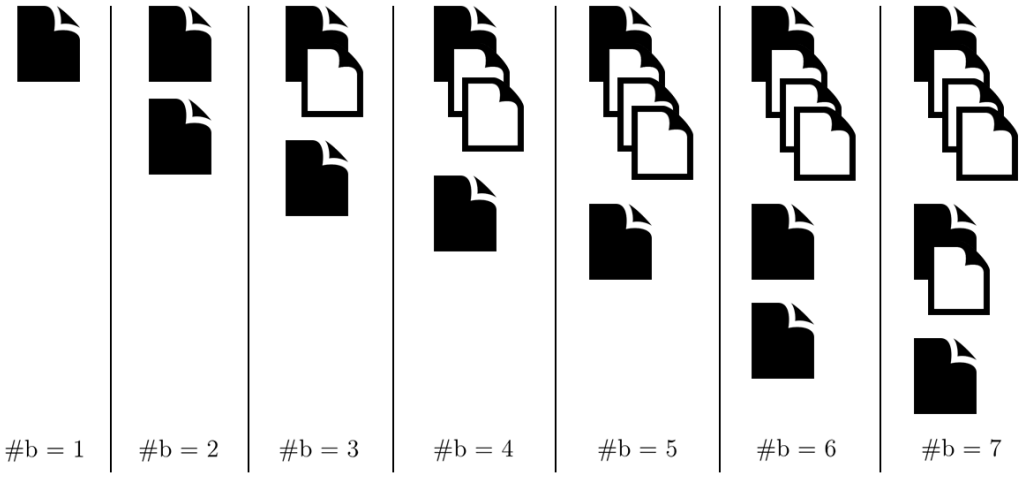

Figure 1: A visualization of the database backup storage method while storing the first 7 backups and maintaining 3 incremental backups after each full backup. Colored papers represent full backup dumps, blank papers represent incremental backups.

Immense storage savings can be accomplished by the demonstrated use of patch files. An unchanged database results in an empty patch file and 100% storage savings. In the unlikely event of the resulting patch file being larger than the original database dump, the backup could simply be stored as a full backup instead of incremental, whereby the diff method would never be able to increase the file size.

To save more space, compression can be applied to each of the database dumps before they are transferred to long-term storage. We chose gzip, as it is a fast, well performing and widely available tool. Gzip compression supports multiple levels from 1 to 9, the lower levels optimizing for fast compression but resulting in larger compressed files, while the higher levels utilize more computing resources and deliver a smaller compressed result.

We conducted an experiment to determine which gzip compression level best suits our needs, by compressing 5 combined database dumps of different size, and compare the storage savings against the duration of the compression. The results are in Figure 2. We are currently using gzip compression level 7, as the required CPU time increases rapidly at level 8 and 9, while the additional savings are limited.

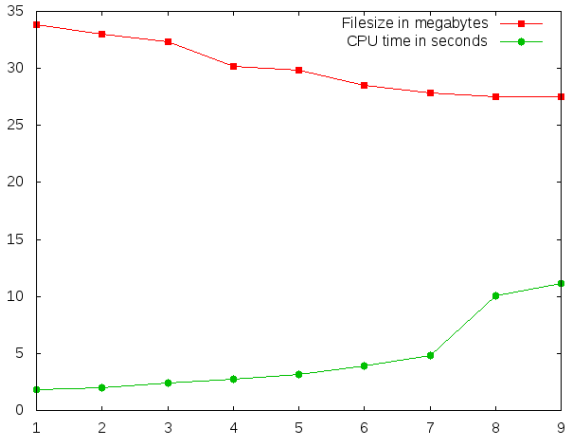

Figure 2: CPU time and compressed file size of representative MySQL databases for all GZIP levels

Database backup is currently implemented for MySQL and its fork MariaDB only. As those databases are the usual default as a website backend, a large part of the market has been covered. However, implementing support for additional database systems should not be difficult, as similar dump tools like MySQLdump exist for different database systems and the storage strategy remains the same.

2.3 Encryption

Another project requirement is robust encryption of both user information and actual backup data. In particular, we want to enable the user to setup encryption in such a way that they themselves are the only one able to decrypt the data, using their password. We have implemented two types of encryption to facilitate two different types of encryption requirements within the application.

2.3.1 Symmetric user data encryption

The first type is straightforward symmetric encryption, which is used for user data such as names, settings, e-mail addresses and other personal details. The purpose of encrypting user information is to limit the impact in case of a security breach of the main database systems: a full database dump should be almost useless without the corresponding encryption keys, which would obviously not be stored on the same server systems that host the database services.

It is often said that one should not roll its own encryption, as proper encryption is complex and a simple mistake can have serious consequences. Robust

and proved symmetric encryption is available in the form of AES, and as such we have used AES 128 bit encryption and constructed a wrapper around it. We have constructed compatible PHP and Python libraries to support symmetric encryption in both the web interface and the Python worker systems. The encryption keys are properly protected and will never appear in any clear form on non-volatile storage.

2.3.2 Asymmetric backup data encryption

Enabling users to protect their files with a password unknown to the application implies encryption in an asymmetric way. Files could be encrypted using a public key, while later on being decrypted with the corresponding private key. Furthermore, if the private key is encrypted with a user's password, that user would exclusively be able to initiate decryption of his files.

Unfortunately, asymmetric encryption is computationally expensive and encrypting all backup data using asymmetric algorithms would not be feasible. Moreover, encrypting data in an asymmetric way increases the file size, leading to higher costs for backup storage.

Hybrid encryption resolves this issue: faster symmetric AES encryption is used to encrypt the actual files, whereafter the AES key is encrypted with the public key of an asymmetric encryption algorithm. If both the (symmetric) file and (asymmetric) key encryption algorithms are secure, the resulting hybrid encryption strategy should be secure as well [8].

Figure 3 displays a schematic overview of the hybrid encryption of a single file, as implemented in the application.

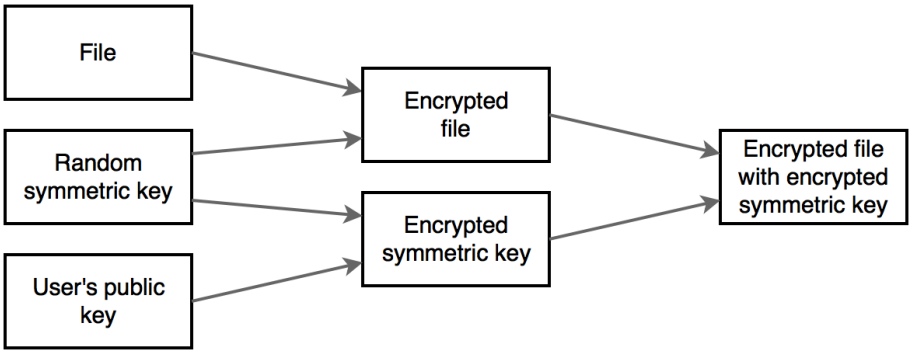

Figure 3: Overview of our hybrid file encryption implementation

We currently offer users the choice of encrypting their private decryption key

with the applications master key or their own password. The first option implies that the application owner is in theory able to decrypt user files as well, while the latter causes all backup data to be lost if the user forgets his password.

In order to supply a backup method to approach files in case of a lost password, a special recovery key is generated when the user activates encryption using his password. The recovery key is a string which is used to encrypt the same private key once again and can be used if the password is lost.

If a user changes his password, the private key can be decrypted with their old pasword, and then be re-encrypted again with the new one. There is no need to change or re-encrypt any of the files during a change of passwords, as the private key remains unchanged.

3 Infrastructure

In the previous chapter, we have explained the backup algorithms and storage facilities in place to perform the backups. Now, we will discuss our infrastructure to execute these algorithms on a large scale and review the ways we have implemented and assembled the separate parts. The application infrastructure consists of six main parts: the main application database, a task distribution queue, processing nodes, file metadata storage, object storage and a backup scheduler. A user interface could be included to simplify management of backups, but is outside the scope of this project. Figure 4 contains a schematic overview of the infrastructure.

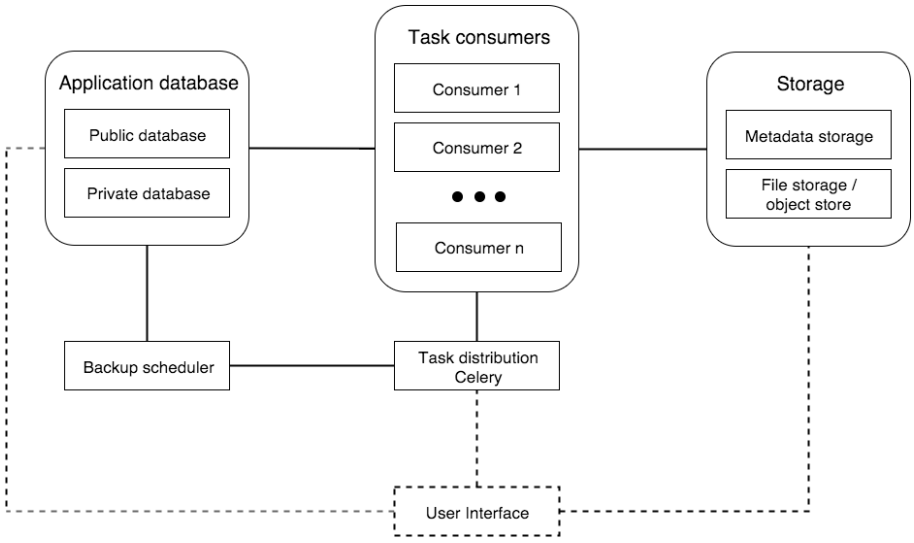

Figure 4: Schematic overview of application infrastructure

3.1 Application database

All non backup related data is stored in the main application database. It concerns website and backup related details, user logins, personal data, settings and more. A MySQL server is currently in place as the application database server, which is replicated twice using master-slave replication to provide for backups. A master-master configuration could lead to conflicts if the same rows are changed simultaneously at different servers, which is why we adopted master-slave replication with automatic promotion of the slave in case of troubles with the master. Hourly dumps of the application database are taken and transferred to a remote system to provide for further backups. Each day, one of the dumps is preserved

for several months in order to protect against corruption and application flaws on a longer term.

For security matters, the application database has been split in two isolated components: a public and private database. The public database contains data required by application components other than the user interface. Examples are backup, restore and logging information used by the interface and worker servers. The private database stores sensitive data, such as user information and session data. The separation allows for additional security of sensitive information, by means of a firewall which limits access to the private database to the website only.

3.2 Task distribution and processing nodes

Application tasks which are expected to run for more than several seconds are handled by processing nodes: dedicated machines handling several application tasks such as backup creation, restoring, constructing downloads and setting up new backup projects.

Distributing the tasks to the worker servers requires a reliable distribution system. Instead of developing yet another tool ourselves, we preferred using an existing application called Celery [9]. Celery is a distributed task queue able to allocate tasks to worker servers using a *broker*, for which we chose RabbitMQ [10].

Celery takes care of several queuing requirements. Tasks are evenly balanced between the available servers and the maximum amount of concurrent tasks can be limited per node. Retrying can be automated in case of task failures, whereby a maximum number of retries and a retry interval can be set. The transport broker RabbitMQ can be installed to multiple servers for redundancy, eliminating any single point of failure during task distribution.

The nodes executing the tasks have been named *processing nodes*. Each node is an instance of Celery, which starts a threaded Python class for every task received. All available tasks have a corresponding Python class capable of processing the steps required to complete the task.

For each task, the unique identifier and type are passed to the celery instance. The worker collects and verifies all other required information - such as task specific arguments, login credentials and encryption keys - using the main (public) database. Furthermore, the backup status, problems and the final result are recorded in the main database during task execution as wel .

An improvement to the local worker storage has been made to increase the

lifespan of the solid state drives installed in workers. During backup creation, each new file is transferred from the remote server to the worker, then verified and - if necessary - transferred to long term storage. The file will be deleted from worker storage immediately thereafter and consequently only exists for a few seconds at most. This practice is rather harmful for the storage cells in the solid state drives, as their lifespan is limited by the amount of write cycles.

To improve disk lifespan and performance, the processing nodes have two types of local storage available: one or multiple solid state drives and a high speed RAM disk of about a gigabyte in size. A RAM disk is a block of random access memory, emulating an ordinary storage volume. The use of a low latency RAM disk highly reduces the number of write instructions for the main storage. As a side effect, it considerably accelerates the backup process and saves a notable amount of time and resources.

3.3 Backup scheduler

The backup scheduler completes a simple yet important task: submitting all scheduled backups as a task to the task distribution component. An hourly cronjob is in place to start the periodic backups as scheduled by clients. The scheduling script is implemented in PHP and analyses the main application database to determine the websites scheduled for backup in the upcoming hour. A backup task is created and inserted into the task distribution queue for each of those websites, whereafter the actual backup will be executed by a processing node.

A considerable amount of verifications is in place to guarantee correct scheduling. Missing backups are to be prevented by matching the run time of the previous schedule iteration to the time of the current iteration. If previous iterations are missing, or an iteration is started twice, the handling should be corrected accordingly. Such glitches could occur due to downtime of the scheduling server or programmatic errors, but happen twice a year anyway during time adjustments due to daylight savings.

3.4 Object storage

The argumentation for the use of an object store has been discussed in the previous chapter. Object stores are easily scalable long term storage appliances, which perfectly suit our needs and are cost effective.

Several software solutions exist to implement an object store, although the fundamental operation is the same. The main components are storage servers,

hosting the actual objects, and proxy servers. The proxies take care of all incoming requests, determining which object on which storage server is referenced on the fly, while carrying out the desired operation. All of the object store servers continuously monitor each other and, in case of a failure, data will instantly be replicated to another server in order to maintain the required redundancy level.

A reliable and redundant object store requires multiple servers in at least three locations, which is expensive to build and maintain. As nowadays many providers of object storage and cloud services exist, we have chosen to use the services of a major Dutch cloud provider for our prototype. If a deployment of our backup application expands to thousands of supported websites, it will gradually become worthwhile to setup and maintain a dedicated object store.

The chosen object store is based on OpenStack Swift [11], part of the open source project OpenStack and supported by many of the worlds largest technological institutions. OpenStack has an HTTP REST API available to manipulate objects in the object store, for which a Python swiftclient is available that integrated easily with our Python based processing nodes.

Encryption of the files takes place on the processing nodes before transfer to object storage. This ensures files are unencrypted on our systems for the shortest time possible - only on our self-managed and secure processing nodes before encryption - and guarantees only encrypted data enters the object store.

3.5 Metadata storage

The revision management system as defined in the previous chapter requires separate storage for metadata and actual files. Metadata concerns backup revision information and all file data other than the files themselves.

Maintaining this data for all backups and websites results in a huge dataset consisting of millions of rows, which continues to grow and does not fit a single server. Our choice for MySQL databases forces us to implement sharding ourselves. The dataset must be split and partitioned over multiple server set and the main application databases keeps track of the set of servers that hosts the metadata database for a specific website. If capacity on a metadata server set falls below a specified threshold, one of the metadata databases it hosts has to be migrated to another set, which is a fairly uncomplicated process.

Replication is done using regular MySQL replication. The metadata servers are grouped in sets: one master metadata database which is replicated to one or multiple slaves.

4 Results

The application has been tested on a number of websites in order to thoroughly verify the operation. We gathered a total of 40 small and medium-sized websites for backup testing with all of them being used for file backup. 32 websites made use of a MySQL database, for which database backups were set up as well. A fair selection of website properties has been made, with candidate websites being hosted on Linux and Windows servers with different types of FTP, SSH and database servers. Some websites with a hosting location in different continents were added as well, to examine backups over high latency connections.

The first run of backups showed a remarkable number of failures which did not occur while testing at the small local scale earlier. Most of the issues were protocol and transfer related, for example including a number of outdated FTP servers without support for the MLSD protocol and firewalls blocking IP addresses of processing nodes after - presumably - too many or aggressive connections.

Furthermore, connection issues appeared, mainly with websites hosted farther away from the processing nodes. The higher latency seemed to increase the number of timeouts and failing file transfers. A solution was implemented by enforcing several retries instead of immediately failing the entire backup in case of a network error.

A new round of backups was initiated after resolving most programmatic and functional issues, which produced considerably better results. After 30 days of backups of all 40 websites, 17 backups had failed unconditionally and 23 backups produced warnings, which corresponds to a failure rate of approximately 1,4% and a warning rate of 1,9%.

All remaining failures were caused by external factors, mostly remote server or network downtime, leading to an unfinished backup. For non-critical websites, those failures would still not require any manual intervention, as a single backup could as well be skipped if the next-day backup completes without issues. Warnings are mainly about files which did exist while the backup directory was indexed, but disappeared before they were actually copied. This would usually not be of importance.

Database backups gave similar results, with a failure rate of 1,4%. Usually if the file backup fails, the database backup fails as well, as both will generally be done at the same time. It is noteworthy that 28% of the database backups contained no changes to the preceding backup, in which case nothing will be stored. Database backups without changes mainly involves not so busy websites

based on a CMS, when content is not frequently updated.

The incremental storage procedures lead to huge storage savings. The average size of an incremental file backup was 3,2% of a corresponding full backup, proving the little amount of changes to the files of the average website. Furthermore, most of the modified files concerned log files, which could be left out of backups by better file selection or the implementation of smart exclusion filters. The average size of an incremental database backup was 3,4%, although real savings are lower due to the scheduled full backup every 25 iterations.

It is harder to determine exact results on our deduplication strategy. With our set of test-websites, we ended up with 12,2% of files having an identical copy of the file for the same website already in storage, leading to storage savings of 7,8%. However, the savings greatly differ per website. For example, several websites were found to have an almost identical copy of the files in a seperate directory for development purposes, and some other websites did not have any duplicate files at all.

Furthermore, if a backup fails halfway, a revision is incomplete and the backup will be marked as invalid. The internal revision numbering continues, however, and the new backup will contain files which were not included in the incomplete backup but are not actually new. Those files are treated as duplicate files as well, which influences the results. Nevertheless, those files would in fact be stored multiple times without deduplication, thus deduplication is useful in case of failed backups as well.

More representative numbers on deduplication should be extracted when the application is in use by much more than 40 websites. At least the current number of identical files seems to be sufficient to justify the additional complexity of deduplication.

The same applies to our restore processes. We performed 100 restores and backup downloads without a failure, but some restores would fail for sure during prolonged use. It could for example go wrong because of faulty permissions on the remote hosting server, but did not in our testing because we anticipated that. Nevertheless, external factors such as permissions or network issues can not be remedied by our application anyway and will always require human intervention.

5 Discussion and conclusion

In this thesis, we have described the implementation and design choices of our web and database backup solution. We have completed a working prototype of the proposed application and improved and verified its correct and expected performance.

In particular, our explicitly defined requirements have been met:

- After setup of their backups, users will only be involved if issues arise or if they actually need to restore a backup. Only 1,4% of backups failed and would alert the user, although even those notifications can usually be ignored if the subsequent backup succeeds.
- All data is stored redundantly by default. Using our current object store, data is stored three times, in two separate physical locations. The main databases and metadata databases are fully replicated in separate locations as well.
- Scalability is secured by making use of easily expandable storage facilities. Creating backups of a million websites using the application should be realistic, as long as the storage systems are expanded in time.
- Users are able to encrypt their backup data in such a way that it can only be accessed when they supply their password.

The application has been tested on a number of websites and has proven to be reliable, with low failure (1,4%) and warning (1.9%) rates and decent reports in case of trouble. Most remaining failures are caused by external factors, although improving to a failure rate of less than 1% will be realistic with additonal improvements on retrying after failures.

Furthermore, the incremental and decuplication procedures save a substantial amount of storage space, thereby lowering operational costs and increasing the speed of backup and restore procedures.

Work on a user interface has begun as well. If a user-friendly and comprehensive user interface were to be completed, people with less technical knowledge would be able to use the application as well and releasing a public application might be an option.

Futhermore, support for additional file transfer protocols and database systems can be implemented. Currently FTP and SFTP are the supported protocols, while MyQL and MariaDB are the supported database systems. However, the market share of additional protocols and database systems is limited.

Lastly, several improvements could be made to the current efficiency of the backup processes. Examples include improved concurrency of transfers and smart file filters, such that log files and temporary data can be excluded from backups automatically to save even more storage space.

References

[1] F. McCown, C. C. Marshall and M. L. Nelson, Why Websites Are Lost (and How They're Somtimes Found), Communications of the ACM, November 2009, p. 141-145

[2] Codeguard, https://www.codeguard.com

[3] ZFS, http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/

[4] GlusterFS, https://www.gluster.org

[5] Ceph, http://ceph.com

[6] MongoDB, https://www.mongodb.com

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel and P. Vajgel, Finding a needle in Haystack: Facebooks photo storage, Proc. of OSDI, 2010

[8] R. Cramer and V. Shoup, Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack, SIAM Journal on Computing, v.33 n.1, p. 167-226, 2004

[9] Celery Project, http://www.celeryproject.org

[10] RabbitMQ, https://www.rabbitmq.com

[11] OpenStack Swift, http://swift.openstack.org

Appendix A

The structure of the four database tables containing all file backup metadata information of a website.

#	Column	Data type
1	directory_id	int(11)
2	directory_path	text
3	directory_name	text
4	directory_type	tinyint(1)
5	directory_permissions	smallint(3)
6	directory_revision_start	mediumint(9)
7	directory_revision_end	mediumint(9)

directories_main

#	Column	Data type
1	file_id	int(11)
2	directory_id	int(11)
3	file_storage_id	int(11)
4	file_name	text
5	file_size_remote	int(11)
6	file_permissions	smallint(3)
7	file_last_modified	datetime
8	file_extension	varchar(128)
9	file_revision_start	mediumint(9)
10	file_revision_end	mediumint(9)

files_main

#	Column	Data type
1	file_storage_id	int(11)
2	file_hash	varchar(32)
3	file_size_local	int(11)

files_storage

#	Column	Data type
1	symlink_id	int(11)
2	directory_id	int(11)
3	symlink_name	text
4	symlink_target	text
5	symlink_revision_start	mediumint(9)
6	symlink_revision_end	mediumint(9)

symlinks_main