DistBack: A Low-Overhead Distributed Back-Up Architecture with Snapshot Support

Thomas Mager, Ernst Biersack
EURECOM, Sophia Antipolis, France
{mager,erbi}@eurecom.fr

***Abstract*—There exist many distributed storage systems tolerating failures of participating nodes. However, they require high amounts of metadata and do not focus on a user's need to easily recover a snapshot of their data. In this paper, we describe DistBack, a distributed back-up system that involves always-on home network gateways with the assistance of a reliable data center. We separate the system into swarms in order to ease monitoring and limit the scope of data requests. DistBack introduces index files which comprise metadata necessary to recover a snapshot. To increase efficiency, we embed small files into these index files. We show that this is reasonable due to the low amount of storage space they account for, which in our case is less than 0.1%. As a result, DistBack requires less metadata to relocate data. It supports snapshot based back-up and provides solutions for storing files of different sizes.**

I. Introduction

Storing data not only locally at home but also at different locations significantly reduces the risk of data loss due to theft or natural disasters. Although data may be very valuable to them, most users underestimate these risks and neglect efforts or costs for an off-site back-up, which is distributed over different locations.

However, today, many users have sophisticated home network gateways (HNG) which include network-attached storage and functionality to seamlessly synchronize data with different user devices at home [1]. Due to the mutual interest in storing a back-up off-site, it is possible to leverage spare resources on such HNGs for free. TotalRecall [2], Glacier [3], and OceanStore [4] are examples of systems using such approach.

Unfortunately, these systems entail a lot of metadata and overhead for checking data availability. They assume data to be organized in abstract blocks of fixed size. Files either consist of multiple blocks or may share a block with other files. Finally, each block is stored on a subset of participants in the network. This results in a lot of metadata and overhead for checking the availability of each block.

Further, existing systems employ block-level *incremental* back-up, so that only differences to a former state of a file system are uploaded [5]. This form of back-up, however, does not support deletion of former back-ups. This is crucial since storage space on foreign nodes is limited. Therefore, after reaching such limit, a user is no longer able to upload recent data. On-site back-up solutions like Time Machine [6] offer *snapshot-based* back-up, supporting deletion of older states. In contrast, we are not aware of any system with distributed snapshot support.

Based on this motivation, the contribution of this paper is as follows:

- A swarm based architecture which uses file level access and reduces necessary metadata to a very low level. Further, it is easy to monitor a swarm.
- We show how to support a snapshot based back-up in a distributed storage system.
- Solutions for efficiently storing files of different sizes in such a system.

The remainder of this paper is structured as follows. First, we present related work in Section II. In Section III we overview our architecture. Section IV illustrates the mapping of back-up snapshots to data structures. The data placement policy is explained in Section V. We address maintenance of stored data in Section VI, followed by a discussion in Section VII concerning files of different sizes. Section VIII provides a recommendation on how to implement the architecture. Finally, we conclude in Section IX.

II. Related Work

For peer-to-peer systems, lots of research has been done with focus on *on-line storage* [7], targeting features usually provided by common file systems. These features include, for example, requirements for latency or consistency of concurrent reads and writes. Since peers are prone to churn, these solutions need additional redundancy and maintenance mechanisms to ensure data availability over time. The redundancy is generated using simple replication, or more sophisticated coding techniques such as Reed-Solomon coding [8], fountain codes [9], [10], or regenerating codes [11]. Unfortunately, these codes come with a trade-off [12]: either higher initial bandwidth requirements and increased storage costs, or higher demand of repair bandwidth later on.

In contrast to on-line storage, for the scenario of *on-line back-up*, Toka et al. show that requirements on the system are relaxed [5]. Writes are solely performed by the data owner and latency is less crucial. Furthermore, a local replica can be used to inject redundancy at minimum bandwidth costs. According to the level of injected redundancy, a certain period without further maintenance can be bridged, so that no data loss is to be feared.

Deduplication [13] is often used in storage systems to avoid storing the same data multiple times. This results in storage and bandwidth cost savings. We do not use this approach to deduplicate files of different users (which entails security concerns [14]), but within a single snapshot as well as across different snapshots.

III. General Architecture Using Swarms

Before we go into the details, we provide an overview of the system architecture. The entities in our system, also shown in Fig. 1, are as follows:

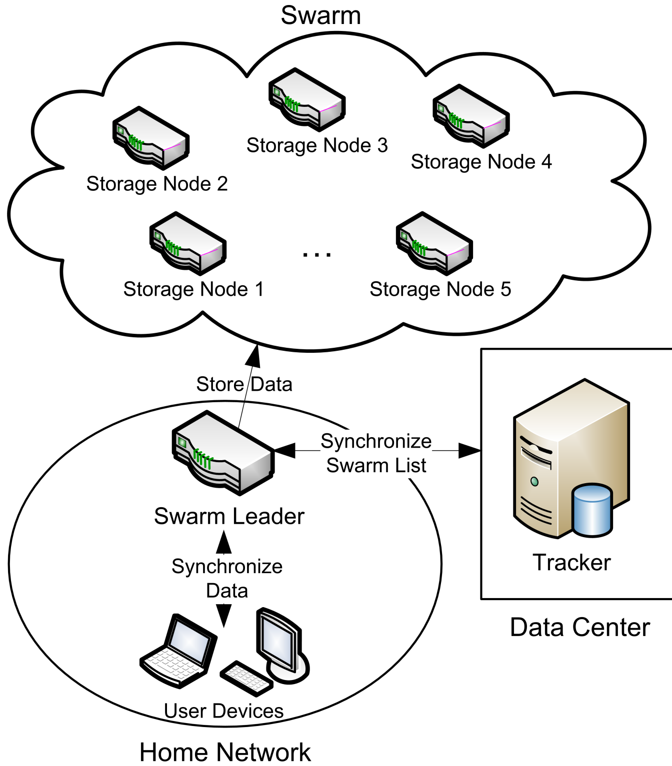

Fig. 1. General Architecture

A. Node

A node is the equivalent of a HNG, that is part of our system. A node has a unique identifier and offers storage space to other nodes. There are two different roles to fulfill for a node:

Swarm Leader: A swarm leader is the owner of a swarm and the data owner of the particular back-up stored in its swarm. It synchronizes data to be backed up with the user devices in the local home network and hence always holds a replica of the data. This way, we unburden user devices of staying connected to the Internet in order to achieve an external back-up. Each node is swarm leader only once.

Storage Node: A storage node is a member of a swarm and consequently stores data related to a swarm leader's back-up. It can be seen as a simple key value store. Typically, a node is storage node in several swarms.

B. Swarm

A swarm is a list of randomly chosen storage nodes. Each swarm leader backs up all its data in such an individual swarm. By design, this means we do not intend to store data in a reciprocative way. Within a swarm, we assign increasing indices to all storage nodes. Further, a swarm can be in one of the following states:

Intact Swarm: The swarm leader is on-line. Whenever a storage node in its swarm leaves, it can upload additional redundancy to a new storage node. This redundancy can be generated using the local replica [5].

Isolated Swarm: The swarm leader is off-line. The state of its swarm needs to be observed by the tracker.

Managed Swarm: The number of available storage nodes in the isolated swarm dropped below a threshold. Without action, data loss is feared. All files stored in the swarm are downloaded and reconstructed in the data center. Only in this case, the user needs to download its back-up from the data center.

C. Tracker

The tracker as a central instance within a data center has several responsibilities, described in the following.

Track Nodes: The tracker keeps track of the availability of all nodes in our system. Therefore, the nodes occasionally send heartbeat messages to the tracker. If several consecutive heartbeat messages are missing, a node is considered to be off-line. The tracking of nodes also includes the mapping of node identifiers to the currently assigned IP address of a node. Nodes can query this information in order to connect to swarm members.

Store List of Uploaded Snapshots: For each swarm leader, we store a list of previously uploaded snapshots on the tracker. This includes the date of a snapshot as well as a reference to the corresponding index file (explained in the following section).

Observe Swarms: It may happen that a swarm leader is off-line for a longer period. To protect its back-up, a tracker needs to identify which swarms are prone to data loss. As mentioned above, these swarms are set to the *managed* state.

IV. Snapshots Yielding Index Files

By **snapshot** we denominate a state of a folder structure with all its files at a particular point in time. In our system, each snapshot consists of the following (also shown in Fig. 2):

- The **file set**, which is the set of all files that are included in the snapshot. Each file in the file set has a unique **file identifier**. The file identifier is deterministically derived by hashing the file contents. This ensures that the file identifier for the same file will remain the same in later snapshots, given the file contents do not differ. In general, only a small subset of files is expected to change from one snapshot to another [15]. Since we are able to map identical files to different snapshots, we avoid storing duplicates of files that are already present in a swarm.
- A single **index file**, which holds all metadata necessary to reconstruct a snapshot. In particular, this includes all file identifiers, the folder structure, and file metadata such as filename and modification time. As we will see in Section VII, we can also profit from embedding small files into the index file.

V. Data Placement

In this section, we explain how fragments of files are generated and where they are placed subsequently. The data placement strategy impacts not only the amount of metadata

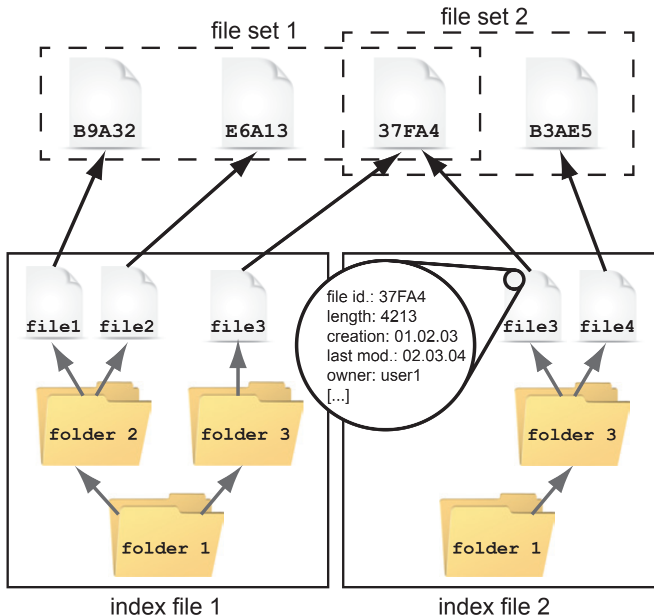

Fig. 2. Example of Two Index Files Having One File in Common

necessary to relocate data, but also how to perform maintenance in the system (see Section VI).

We split a file of size S_f to be stored in a swarm into k equally sized fragments T_i with an index $i \in \{1, \ldots, k\}$. Hence, each fragment has a size of $\lceil S_f/k \rceil$. Using erasure codes (such as Reed-Solomon [8], or fountain codes [9], [10]), we can generate h additional fragments $T_i, i \in \{k+1, \ldots, h\}$. As a property of erasure codes, any k out of the $k+h$ different fragments will be sufficient to reconstruct the original file content later on.

For data placement, we group fragments of all files $f_1 \ldots f_n$ in the swarm by their index i to a **redundancy stream**. Consequently, such a redundancy stream RS_i is defined as:

$$RS_i = T_{i,f_1}, T_{i,f_2}, \ldots, T_{i,f_n} \tag{1}$$

The size of a redundancy stream depends on the total amount of data S_t for the back-up and is determined by $S_{RS} = S_t/k$.

On each storage node within a swarm we place a different redundancy stream. This placement policy transfers the above-mentioned property of erasure codes on nodes: any k out of $k+h$ storage nodes are sufficient to recover all files stored in a swarm. Since all storage nodes in the swarm hold a fragment of each file, we spare additional metadata or look-ups to relocate data corresponding to a file. The knowledge of a single file identifier is sufficient to query different fragments of the particular file.

VI. Maintenance

This section focuses on the process of maintaining a back-up alive in the swarm. The swarm leader performs this task, given it is not off-line for a longer period. Otherwise, as mentioned before, a replica is downloaded to the data center.

A. The Need for Maintenance

In our system, we need to consider failures of nodes. These failures can be divided into two categories. If a node is only temporarily unavailable for other nodes, such a failure is denoted as a **transient failure**. For a **permanent failure**, the node left the system forever and data stored on the node is lost permanently. Unfortunately, it is impossible to distinguish these failures from outside because both types of failures have the same characteristics, while it is unknown how long a transient failure of a node lasts. In contrast to transient failures, permanent failures entail data loss in a swarm and, consequently, require the upload of additional redundancy.

B. New Redundancy and Reintegration

In our system, we only add new redundancy, instead of restoring previously uploaded redundancy that is temporarily unaccessible. According to the data placement policy described in Section V, this means a new redundancy stream is uploaded to a new storage node. After the complete upload, the storage node is included in the swarm, which therefore grows in size over time.

This implies that we are always able to *reintegrate reappearing nodes*. As investigated by Wheatherspoon et al. [16] this can significantly reduce the maintenance costs of the system: in the long term, only permanent failures trigger an upload of additional redundancy.

C. Redundancy Needed

We need to inject more redundancy in the system in order to bridge a time period without further maintenance by the swarm leader. The amount of necessary redundancy also depends on the average lifetime τ of nodes in the system. Assuming node lifetime values to be independent and exponentially distributed, we can calculate the probability p_{dur} that at least k out of $k+h$ nodes in a swarm remain intact within the time period t_{iso} [5]:

$$p_{dur} = \sum_{i=k}^{k+h} \binom{k+h}{i} (e^{-t_{iso}/\tau})^i (1 - e^{-t_{iso}/\tau})^{(k+h)-i} \tag{2}$$

Given the average availability α of storage nodes in a swarm, new storage nodes need to be added as long as there are less than $\lceil (k+h) \cdot \alpha \rceil$ storage nodes on-line. Fig. 3 shows an exemplary evolution of a swarm list at four points in time.

VII. File Handling

In the Section V we decided to send fragments of size $\lceil S_f/k \rceil$ to storage nodes. Unfortunately, in practice, this approach is accompanied by drawbacks resulting in inefficiency. As shown in [17] and [18], it can be advantageous to store very small files close to their metadata. In our scenario, this means we store small files embedded in the index file. As a trade-off, such small files are excluded from deduplication over several snapshots.

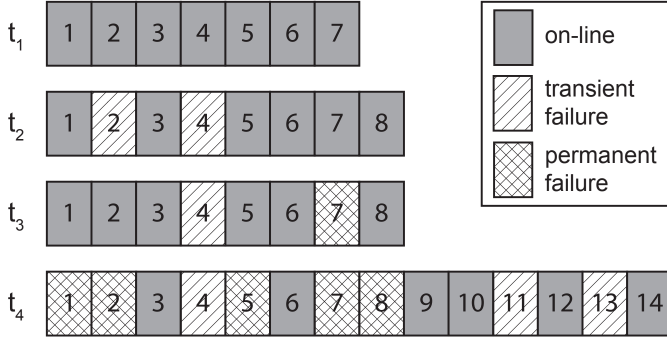

Fig. 3. Evolution of a Swarm List Due to Maintenance. t_1: The initial upload to $k+h = 4+3$ storage nodes; t_2: Two storage nodes are off-line. Assuming $\alpha = 0.8$, $\lceil (4+3) \cdot 0.8 \rceil = 6$ storage nodes need to be on-line, thus, one is added to the swarm; t_3: One storage node permanently fails. Since there are still 6 storage nodes on-line, nothing needs to be done; t_4: Advanced state after several transient and permanent failures.

In order to determine the limit file size at which a file should be embedded in the index file, we define functions for the local storage costs on a single storage node for placing a single file of size S_f. For embedded files, these costs depend on the number of snapshots v in which a file is used in and on the overhead S_{to} within the index file. This constant overhead is used for storing an additional file entry with all its metadata. The costs S_{emb} for an embedded file are defined as follows:

$$S_{emb} = v \;\cdot\; \frac{S_f + S_{to}}{k} \tag{3}$$

For the case where we store each file individually in the swarm, the costs are divided into two parts. The first part are the singular costs for storing a fragment of the file on a storage node. These include the costs S_h for the file identifier. Additionally, there is some constant overhead S_{so}, depending on the storage engine used on storage node side. The second part are the costs for the reference in the index file, which needs to be included for each snapshot. At this point, we also need to store the original file size of the stored file in S_l. This information is needed to crop a decoded file to its actual file size again. The costs for an individually stored file finally are:

$$\begin{aligned} S_{ind} = {} & \frac{S_f}{k} + S_h + S_{so} \\ & + v \;\cdot\; \frac{S_h + S_l + S_{to}}{k} \end{aligned} \tag{4}$$

To find the break even point for the costs of embedded respectively individual files, we identify the costs for storing an individual file with the costs for storing an embedded file and solve for S_f:

$$\begin{aligned} S_{emb} &= S_{ind} \\ v \;\cdot\; \frac{S_f + S_{to}}{k} &= \frac{S_f}{k} + S_h + S_{so} \\ & \quad + v \;\cdot\; \frac{S_h + S_l + S_{to}}{k} \end{aligned} \tag{5}$$

$$S_f = \frac{S_h \cdot (k+v) + k \cdot S_{so} + S_l \cdot v}{v-1} \tag{6}$$

Since we do not want to make an assumption on the number of snapshots stored by a user, we look at an unlimited number of snapshots:

$$\lim_{v \to \infty} \frac{S_h \cdot (k+v) + k \cdot S_{so} + S_l \cdot v}{v-1} = S_h + S_l \tag{7}$$

However, this only provides a lower bound considering storage space efficiency. There are more reasons to embed small files. This includes computational overhead, which depends on the erasure code used, and the additional bandwidth used for sending and requesting small fragments. Additionally, splitting files into fragments globally increases fragmentation, thus, leads to increased disk head movement. At worst, this can lead to bottlenecks which might increase the time needed for uploading redundancy to the system. As investigated in [19] and [20], this needs to be avoided. Consequently, this means a system design may need to disregard strict storage space efficiency and, instead, embed more files.

Here we can benefit from the fact that the total storage overhead in the system depends on how much storage space is occupied by smaller files in general. Based on data of 2004, Agrawal et al. show in a study that typically, file systems contain a lot of small files, but the majority of stored bytes are found in increasingly larger files [17]. They determine that the mean file size in file systems grows each year. Unfortunately, there is no such comprehensive study concerning file size distributions of current file systems. We asked the authors of Wuala [21], a widely used cloud storage provider, for statistics on their file size distribution. We received statistics on all incoming files for the month of October, 2012. Fig. 4 shows the corresponding CDF of used storage space by file size.

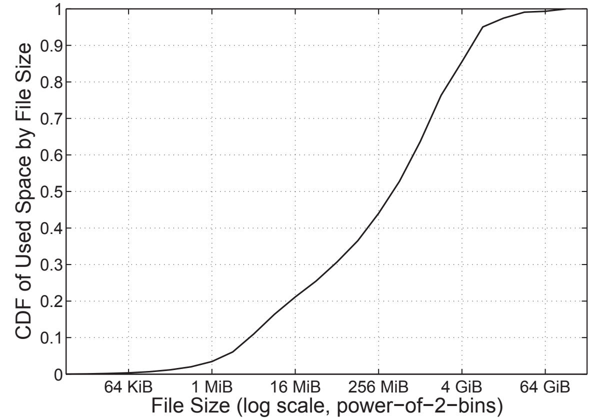

Fig. 4. CDF of Used Space by File Size

As can be seen the amount of storage space used for smaller files is fairly low. In fact, files smaller than 128 KiB account for less than 1% of the total storage space used, while files up to 16 KiB occupy less than 0.1%.

VIII. Implementation

This section provides further information about an implementation of the proposed architecture.

A. Joining the System

The first time a node joins the system, it needs to create an initial swarm. This is done in the following steps:

1) The node contacts the tracker.
2) The tracker observes the node until it shows a minimum level of availability.
3) The tracker includes the node in a list of available storage nodes.
4) The node asks the tracker for a random list of other nodes to build its individual swarm.

After these steps the upload of data to other nodes in the swarm is performed like described in the following section.

B. Uploading a Back-up Snapshot

To keep the back-up in the swarm in synchronization with the local data stored on a swarm leader, a new snapshot needs to be created. We avoid orphaned file references by dividing the upload into the following sub-phases:
Upload files missing in the swarm: We do not expect all files to change between two successive snapshots. To skip files already uploaded as part of previous snapshots saves storage, bandwidth, and processing resources.
Upload a snapshots index file: The upload of an index file finalizes the upload of a new snapshot. All files previously uploaded can be referenced afterwards using this file.
Delete exclusive files of oldest snapshot: As soon as a swarm leader runs out of storage in its swarm, we want to free the storage occupied by the oldest snapshot. Therefore, we delete all files that are referenced in the oldest, but in no newer snapshot anymore. We can identify these files using simple reference counting [22].
Delete index file of oldest snapshot: By deleting the index file of our oldest snapshot we complete the deletion of our oldest snapshot.

C. Security

This section explains our usage of state of the art technology to ensure data confidentiality, authentication, and data integrity.
Encryption of Data: To preserve data confidentiality we encrypt data using AES 256-bit before uploading fragments of our data into the system. This ensures that nobody, except the data owner, can access the original content of a back-up. To derive an encryption key, we use a key derivation function [23] with the user's log-in credentials in combination with password strengthening [24]. This encryption key is the same for all files, including index files. Without the knowledge of the log-in credentials it is therefore not possible to decrypt any data stored in the system.
Authentication of Swarm Leaders: A swarm leader needs to update its swarm list and upload fragments to storage nodes. To prevent attackers modifying a foreign swarm list or data stored in foreign swarms, we use the tracker as a central authentication instance. The user credentials with a unique node identifier are used to create an authenticated session for the swarm leader at the tracker. Before accepting data to be stored, storage nodes check the authentication state using the Kerberos protocol [25].
Checking Data Integrity: Since the swarm leader has a local replica available, we use a conventional challenge-response protocol [26] to check a fragment's integrity on a storage node. Only if the corresponding fragment is available on the storage node, it is able to reply with the correct response.

D. Data Structure for the Index File

We use a compressed tar-archive to store all necessary information of an index file. The folder structure of a back-up snapshot can be preserved by copying the original folder structure to this archive. For small files we also copy the complete files in this folder structure. For larger files we only create a dummy file with the same name at the same location. In this dummy file, however, we store the actual file size and a reference which allows us to query the relevant file content from the storage nodes later on. The index file can be uploaded to storage nodes as a common file.

E. Parameters for Erasure Coding

In our implementation we use Reed-Solomon coding, since it is an optimal code and therefore uses the least amount of storage. Further, for every erasure code some metadata is required for the decoding process. For Reed-Solomon, this is a simple increasing integer value. Since we already have increasing indices for storage nodes, we can merge this information and spare additional metadata.

The choice of parameter k has several consequences. According to Formula 2, a higher value for k reduces the necessary redundancy level in the system. On the other side, the metadata overhead increases as expressed by Formula 4. We also need to consider that the whole file tree needs to be scanned for the creation of a single redundancy stream, resulting in higher disk head movement. This needs to be done more often for a high k. These factors taken into account, we recommend a high, but not extremely high value for k, which is 100.

F. Behaviour Dependent on File Size

For our implementation, we define three classes of files which are handled differently:
Small files: Files with a size up to 16 KiB are embedded into the index file (see Section IV) and need no further treatment. They will be encrypted and encoded together with the index file. Note that all small files are included in each snapshot. Due to the small amount of storage they occupy (see Section VII), this seems acceptable.
Medium files: After encryption, all files from 16 KiB to 1 MiB are erasure coded using the whole file as coding block. This results in fragments of size $\lceil S_f/k \rceil$ which entails a maximum overhead of $k - 1$ Bytes per file.
Big files: Since erasure coding is performed in memory, we cannot perform it on files with arbitrary size. This is why we need to switch to an interleaving scheme, as seen in Wuala [18]. To decrease the resulting storage overhead, files exceeding 1 MiB are broken into coding blocks of size $b = 100$ KiB. On each coding block erasure coding is performed. Therefore, we obtain fragments with a size of $\lceil S_f/b \rceil \cdot b/k$, which entails a maximum overhead of 10% for files slightly larger than 1 MiB. For larger files this overhead becomes negligible.

According to this arrangement, a storage node receives fragments of a size down to 164 Bytes. Since we send fragments of different files to the same storage node (see Section V), we do not have to open new connections for each fragment. Further, we want to avoid wasted storage space on

a storage node due to block alignment in the file system. Hence, we place fragments smaller than 256 KiB [27] in a local lightweight database, using the file identifier as key. Typically, databases provide compaction functionality, so that fragmentation due to deleted fragments can be easily coped with. Fragments bigger than 256 KiB are stored as files in the file system using the file identifier as filename.

G. Downloading a Back-up Snapshot

Similar to the upload of a snapshot we can separate the download of a snapshot into phases. In the first phase we retrieve the index file while in the second phase we retrieve all files referenced in the snapshot.
Retrieve index file: We can recover the file identifiers for an index file by sending a query to the tracker. Afterwards, the swarm leader needs to query all necessary fragments in the swarm related to the index file. After all relevant fragments are downloaded from the swarm, the index file can be decoded and decrypted using the user credentials. The decrypted index file allows us to reference all the files included in the snapshot.
Download all files referenced: Using the file identifiers stored in the index file, the whole file set of a snapshot can be downloaded. To do this, the tar-archive is extracted to restore the folder structure. Note that this extraction includes already small files which are stored within the index file. For medium and big files, all necessary fragments can be requested in the swarm using the deposited file identifiers. After decoding, the files can be decrypted using the key derived from the user credentials.

The system design also supports recovery of single files within a snapshot. In this case, it is enough to download the index file and the desired files subsequently. For a managed swarm, the data center needs to download all snapshots, which covers all files stored in a swarm. Storage nodes can list all these files so that no decryption of index files is necessary in the data center.

IX. Conclusion

In this paper we introduced our distributed back-up architecture DistBack. In our scenario, we benefit from a local replica that allows us to add additional redundancy at low costs. Further, we reintegrate reappearing nodes so that in the long term the system only suffers from permanent failures.

Based on this background we introduced swarms which are easy to monitor and therefore only impose little management effort on a data center. In combination with our data placement policy, we reduce metadata necessary for data localization to a very low level. Using index files, we are, to the best of our knowledge, the first to provide a distributed back-up with support for snapshots. We saw that it is reasonable to embed small files into these index files, due to the low resulting storage overhead. Along with these benefits, we provided instructions for an implementation of the proposed architecture.

Acknowledgement The research leading to these results has received funding from the European Commission's Seventh Framework Programme (FP7 2007-2013) under grant agreement n. 258378.

References

[1] Free SAS. (2013) Freebox revolution. [Online]. Available: http://www.free.fr/adsl/freebox-revolution.html

[2] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker, "Total recall: system support for automated availability management," in *USENIX NSDI 2004*.

[3] A. Haeberlen, A. Mislove, and P. Druschel, "Glacier: highly durable, decentralized storage despite massive correlated failures," ser. NSDI'05. USENIX Association, 2005, pp. 143–158.

[4] J. Kubiatowicz and et al., "Oceanstore: an architecture for global-scale persistent storage," *SIGPLAN Not.*, pp. 190–201, 2000.

[5] L. Toka, P. Cataldi, M. Dell'Amico, and P. Michiardi, "Redundancy management for P2P backup," in *IEEE INFOCOM 2012*.

[6] Apple Inc. (2013) Time machine. [Online]. Available: http://www.apple.com/support/timemachine/

[7] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," in *ITCC 2005*.

[8] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*. Wiley-IEEE Press, 1999.

[9] M. Luby, "Lt codes," in *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, 2002.

[10] A. Shokrollahi, "Raptor codes," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551 –2567, 2006.

[11] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," in *IEEE INFOCOM 2007*.

[12] L. Pamies-Juarez and E. Biersack, "Cost analysis of redundancy schemes for distributed storage systems," *CoRR*, 2011.

[13] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in *Proceedings of the 4th ACM international workshop on Storage security and survivability*, ser. StorageSS '08. ACM, 2008.

[14] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *ACM CCS 2011*.

[15] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads," in *USENIX ATC*. Berkeley, CA, USA: USENIX Association, 2008.

[16] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient replica maintenance for distributed storage systems," in *NSDI 2006*.

[17] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," *Trans. Storage*, vol. 3, no. 3, Oct. 2007.

[18] Thomas Mager, Ernst Biersack, and Pietro Michiardi, "A measurement study of the Wuala on-line storage service," in *IEEE P2P 2012*.

[19] V. Venkatesan and I. Iliadis, "Effect of codeword placement on the reliability of erasure coded data storage systems," IBM Reseach, Tech. Rep., 2012.

[20] F. Giroire, J. Monteiro, and S. Perennes, "P2P storage systems: How much locality can they tolerate?" in *IEEE LCN 2009*, Oct. 2009.

[21] LaCie AG. (2013) Wuala cloud storage service. [Online]. Available: http://www.wuala.com

[22] D. Bulka and D. Mayhew, *Efficient C++: performance programming techniques*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[23] B. Kaliski. (2000) RFC 2898 - PBKDF2 key derivation function. [Online]. Available: http://www.ietf.org/rfc/rfc2898.txt

[24] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password strength: an empirical analysis," in *IEEE INFOCOM 2010*.

[25] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Usenix Conference Proceedings*, 1988.

[26] Y. Deswarte and J.-J. Quisquater, "Remote Integrity Checking," in *Sixth Working Conference on Integrity and Internal Control in Information Systems (IICIS)*. Kluwer Academic Publishers, 2004.

[27] R. Sears, C. Van Ingen, and J. Gray, "To BLOB or not to BLOB: large object storage in a database or a filesystem," Microsoft Research, Tech. Rep., 2006.