Introduction

The aim of this essay is to provide an infrastructural design for a large-scale distributed filesystem aligned to the requirements set out by Onefile. We shall construct our proposal by drawing on technologies from a variety of sources, including traditional distributed filesystems, peer-to-peer systems, and Cloud computing. We intend our end-result to have consistency semantics that are no worse than those offered by the most popular distributed filesystems in use today, whilst providing enhanced opportunities for scalability, performance, and fault-tolerance.

More specifically, we shall be referring to the Andrew File System (AFS) [1] as the baseline for our client–server setup, and taking advantage of Coda's improvements for increased availability, including disconnected operation [2]. We shall argue for the use of a distributed hash table (DHT) as a network overlay providing location transparency of the storage servers, harnessed through a lookup protocol such as Chord [3]. The successful deployment of peer-to-peer file-sharing systems has demonstrated the suitability of such an approach for Internet-scale applications [4]. Then, we shall show how applying a cryptographic hash function, such as SHA-2, over the files' contents can implicitly achieve several of our goals, including single-instance storage (alternatively called data deduplication) and load balancing, as exemplified through systems such as CFS [5] (which is built on Chord) and Venti [6].

Design

Filesystem Structure

At a conceptual level, we shall mirror the filesystem structure that users have grown accustomed to in today's mainstream consumer operating systems (as implemented, for example, in the UNIX File System (UFS) [7], FAT, and NTFS). We shall assume a tree of directories organized in a hierarchical structure, with each directory optionally containing a number of child subdirectories and/or files. Each directory/file has associated metadata, including permissions (access rights) for users and groups, and a timestamp identifying when it was last modified. Files are split into a sequence of blocks, such that each block may be independently stored at an arbitrary location.

Venti [6] sets out a block-based storage system that can store three types of blocks:

- **Directory blocks** are identified by their unique directory path, and contain the metadata for their child subdirectories and files organized as a table. The subdirectory entries would point to other directory blocks; file entries would point to either data blocks (if the respective file's contents fit into a single block) or to pointer blocks.
- **Pointer blocks** are introduced to permit scalable and load-balanced storage for large files. Each pointer block would typically consist of a sequence of pointers to the data blocks that collectively constitute the file. In the case of very large files, pointer blocks may point to other pointer blocks in a hierarchical manner; the depth of this hierarchy would scale logarithmically with the size of the file. (This concept is analogous to inode pointer structures in UFS [7].)
- **Data blocks** contain the raw data for a contiguous portion of the file.

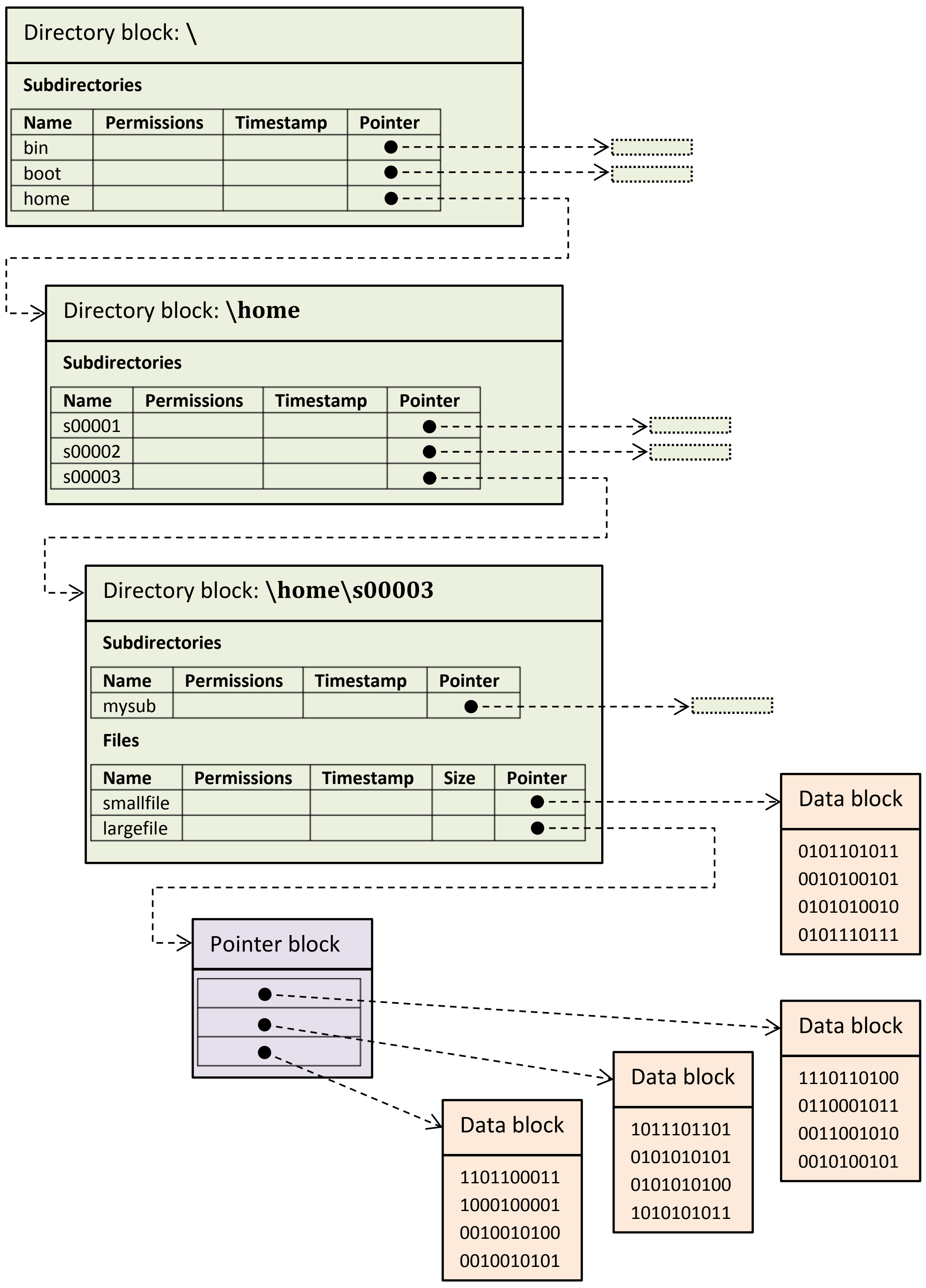


Figure 1. Sample hierarchical directory structure for \home\s00003. The directory contains two files: smallfile and largefile. The contents of the former fit into a single data block, whereas the latter's need to be split across three data blocks through the indirection of a pointer block.

Addressing Scheme

One aspect of traditional filesystems that may serve as a drawback in large-scale distributed scenarios is their use of an extrinsic addressing scheme. The address selected for storing each block is determined arbitrarily with respect to the identity of the block; there is no implicit relationship between each block and its address. As a consequence, explicit bookkeeping mechanisms need to be employed for keeping track of the location where each block is stored. On distributed filesystems, these are made available to clients through a lookup server, which can quickly become a bottleneck (as well as a single point of failure, unless replicated). AFS mitigates the issue by spreading out this responsibility over multiple volumes [1]; however, such an approach results in a skewed load-balance, since volumes containing a lot of frequently-accessed files can still get overloaded.

One of the main insights of DHT-based filesystems is that a block's address may be derived implicitly from its identity:

- The identity of a directory block would correspond to its full directory path [8].
- The identity of a data block would be its entire raw binary contents [5], [6].
- The identity of a pointer block would be the concatenation of its sequence of pointers [6].

In order to obtain a concise representation of this identity, we shall apply a hash function over it. Cryptographic hash functions, such as the SHA family, are particularly well-suited for this end, since they provide a deterministic means of calculating the hash for any given block with a practically-negligible risk of collisions [6], [9]. (There have been no reported collisions on SHA-2 to date.) Being one-way functions, it is computationally infeasible to deduce the original data from just its hash [10].

Cryptographic hash functions provide a uniform distribution over the range of generated hashes, and ensure that slight changes in the original data will result in a completely different hash, a property known as the avalanche effect. Therefore, by deriving each block's address from its computed hash, we would not only ensure that these addresses can be determined intrinsically (without requiring lookup servers), but also that the blocks get approximately evenly distributed across the whole address space [5], [6].

Thus, in order to map this addressing scheme onto our filesystem structure, we would simply need to substitute hashes for pointers. Hashes for subdirectories do not need to be stored, since they may be derived from their full directory path at runtime.

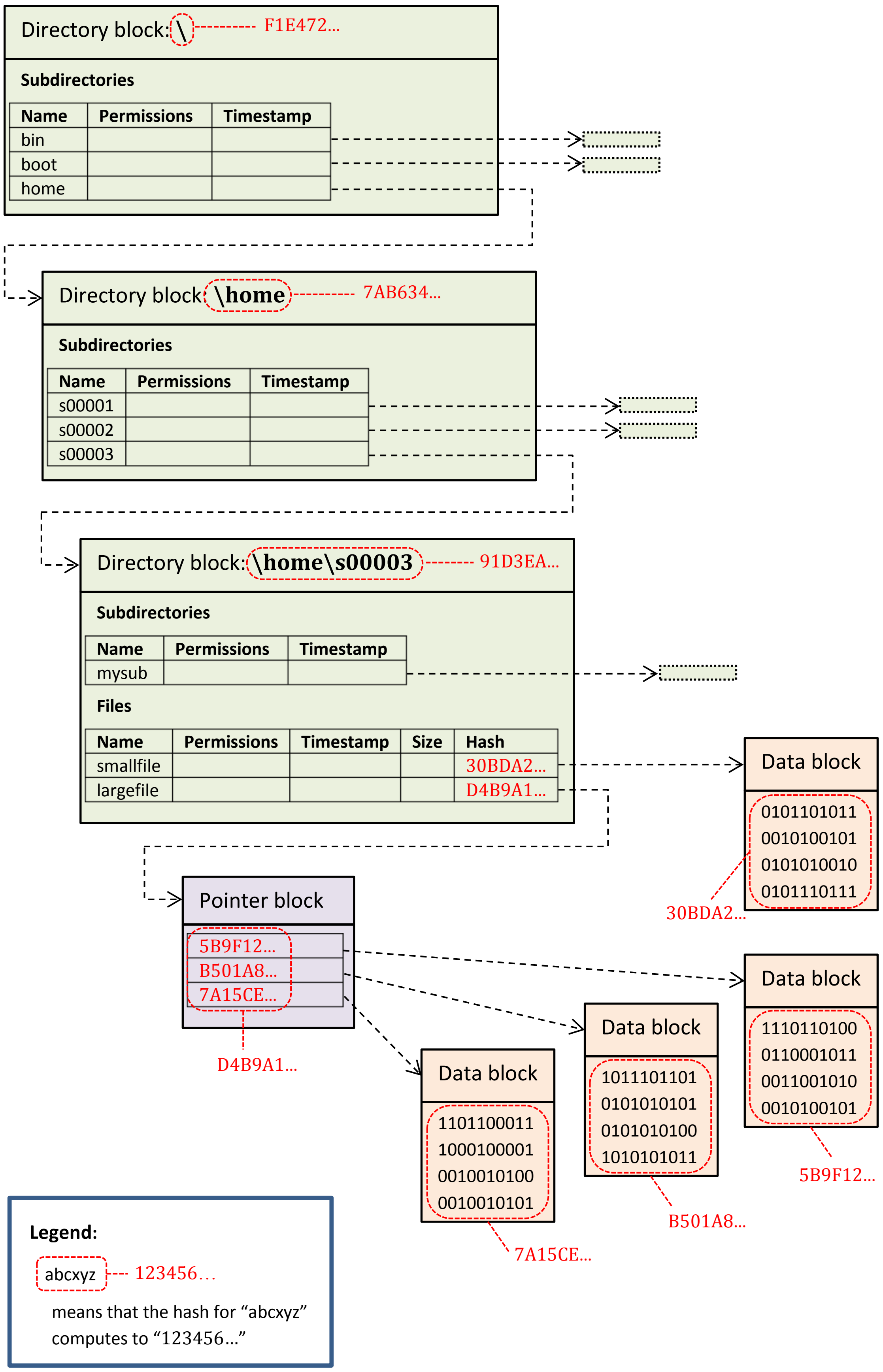


Figure 2. Same directory structure as in Figure 1, but with hashes introduced instead of pointers.

Single-Instance Storage

Another consequence of using a deterministic hash function to derive each data block's address from its contents is that the system implicitly achieves single-instance storage, since all attempts to store the same content would always resolve to the same address. Thus, multiple writes of the same data block become idempotent [6]. This property extends transitively to pointer blocks representing multiple data blocks, or even hierarchies thereof – a concept originally demonstrated for hash trees by Merkle [11].

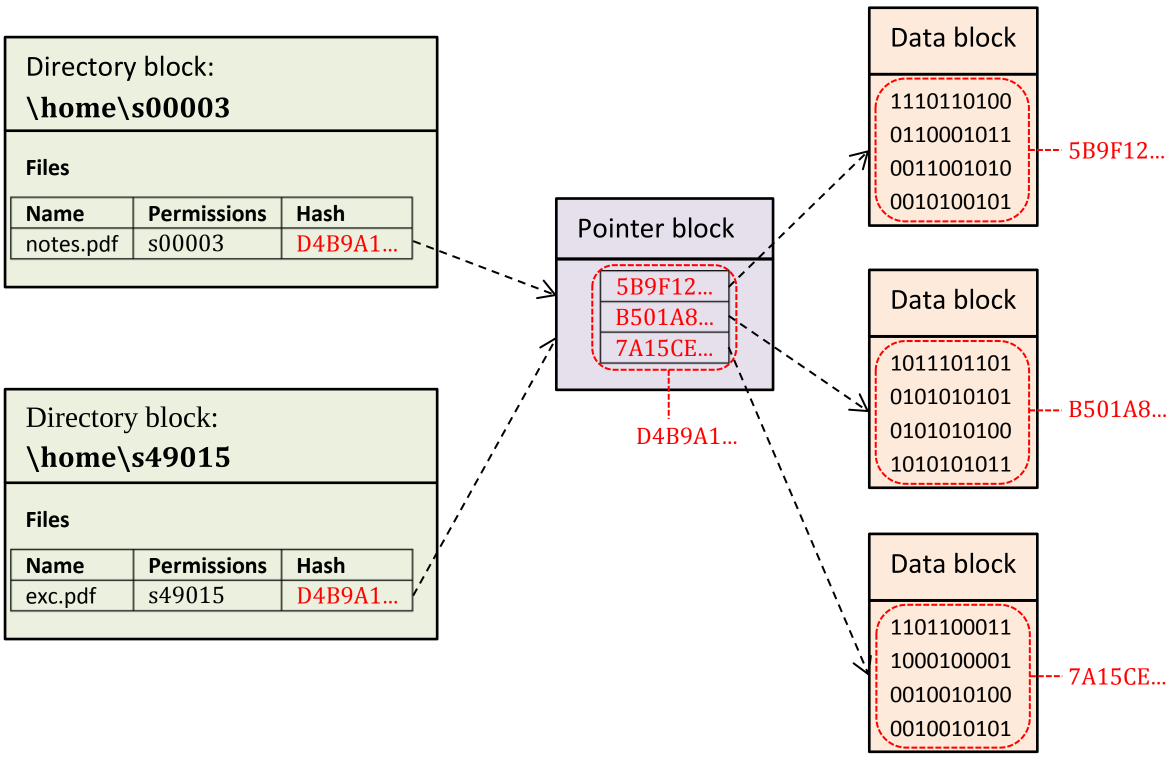


Figure 3. Single-instance storage applied for identical files stored in different directories, belonging to different users.

File metadata, including its name and permissions, is stored within its parent directory, and does not affect the hash. This way, logical instances of files may be accessed and managed separately, even when they point to the same physical data – a behaviour very similar to the concept of hard links in POSIX.

When a file's contents are modified, its hash would also consequently change; this means that the new version of the file would need to be stored at a different address from the old. Thus, all data and pointer blocks become intrinsically write-once [11], and can never become inconsistent.

If a change only affects a small portion of a file, then single-instance storage would still apply for all the data blocks (and, possibly, pointer blocks) that were unaffected; this allows space savings to still be reaped for files whose contents are largely similar but not identical. Space savings would be maximized if one uses Rabin fingerprinting for computing hashes, rather than fixed-size blocks [12], [13].

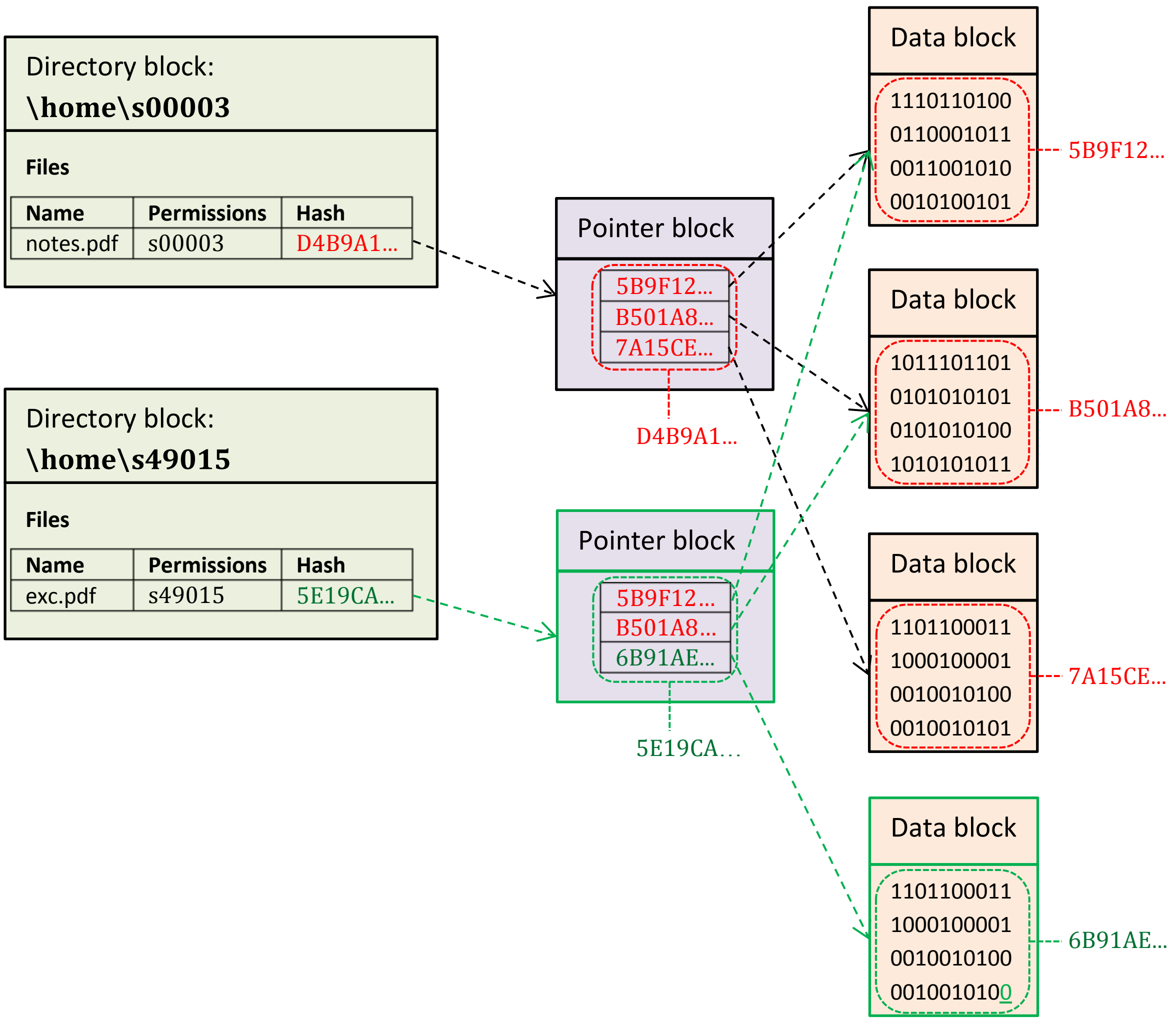


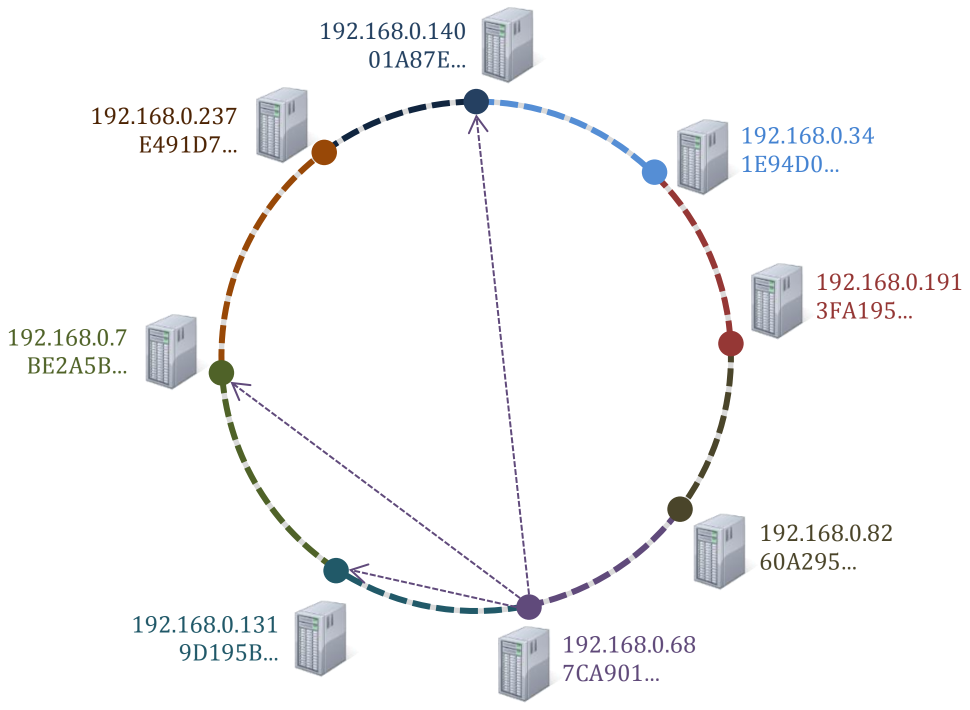
Figure 4. A change to a single bit of the file would result in a completely new hash for the data block, which propagates up to the pointer block, and implicitly requires the file's entry in its parent directory to point to a new address.

After the new data (and pointer) blocks have been written, the new hash for the file needs to be updated in its parent directory. Directory blocks do not share the same write-once property, since their hash is computed from their full path. Thus, any updates to directory blocks would need to be serialized for atomicity.

Storage Servers

Using the protocol defined in Chord [3], one may establish a distributed hash table (DHT) as an overlay network for the storage servers, where each server's identifier hash is computed from its fixed IP address (or, alternatively, its MAC address). Each server would be assigned responsibility for the portion of the address space for which it is the closest successor (that is, the ranges of hashes lying between its own identifier hash and its immediate predecessor's). Due to the random distribution achieved by cryptographic hash algorithms, these portions are roughly equivalent in length, and would remain so even as servers are added to or removed from the network, provided that the network is large enough [14].

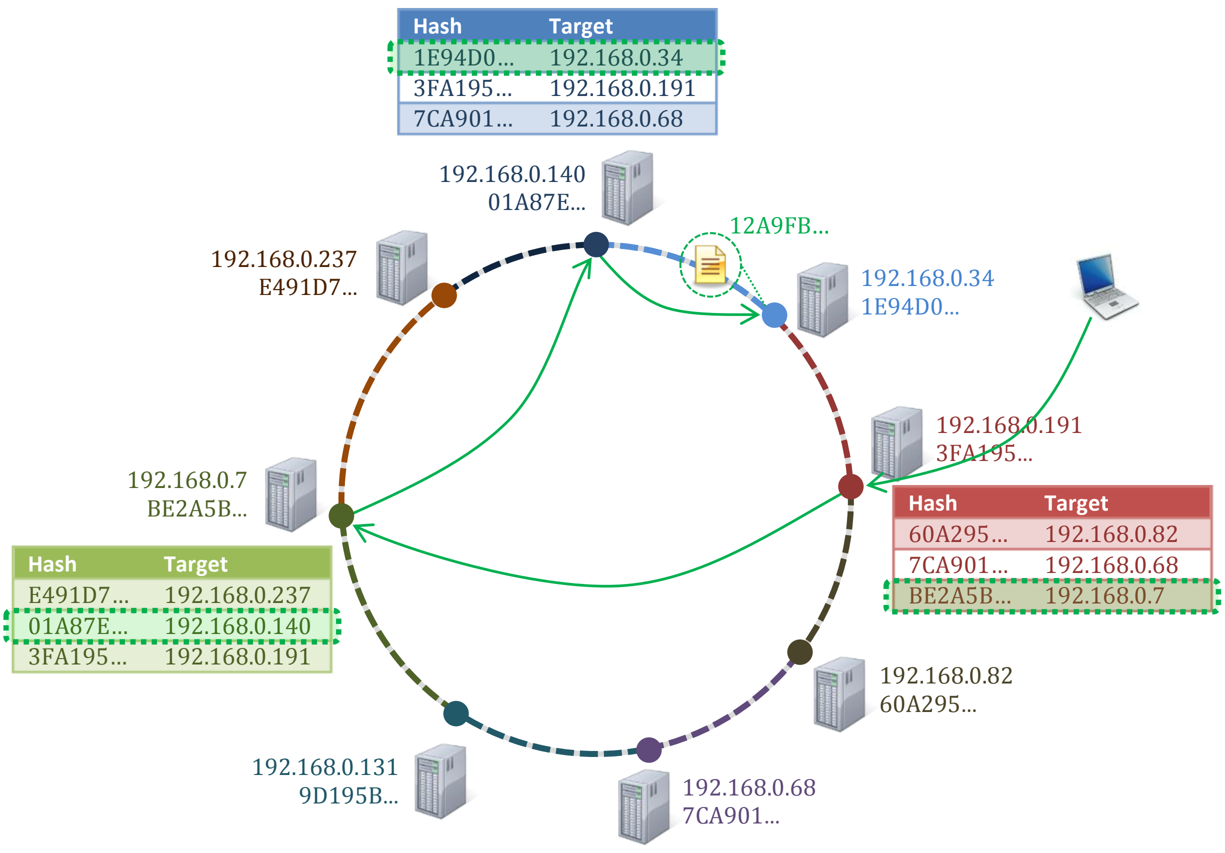
Chord [3] expects each server to keep a record of a small number of other servers participating in the network, situated at exponentially-increasing distances along the hash space; this is known as the finger table.



Hash	Target
9D195B...	192.168.0.131
BE2A5B...	192.168.0.7
01A87E...	192.168.0.140

Figure 5. A Chord ring [3] for eight servers, showing the finger table for the server 192.168.0.68. Each server's finger table would contain $\log_2 N$ entries, mapping hashes to physical addresses of other servers (where N is the size of the network).

Whenever a client needs to find the server responsible for a particular hash in Chord [3], it can start off by sending the request to any server at random. The server would check whether it is responsible for the said hash. If it is, it would just report success to the client. If not, it would consult its finger table and forward the request to the server whose hash is the closest predecessor. This process, which at least halves the remaining address space at each step, is bound to complete in $O(\log_2 N)$ steps, where N is the total number of storage servers.



Hash	Target
1E94D0...	192.168.0.34
3FA195...	192.168.0.191
7CA901...	192.168.0.68

Hash	Target
E491D7...	192.168.0.237
01A87E...	192.168.0.140
3FA195...	192.168.0.191

Hash	Target
60A295...	192.168.0.82
7CA901...	192.168.0.68
BE2A5B...	192.168.0.7

Figure 6. The path taken by Chord [3] to resolve a request for the block having hash "12A9FB...". The client initially contacts a random server, which forwards to request to the closest server in its finger table. This is repeated until the request reaches the server having the succeeding hash, "1E94D0...", which therefore holds the requested block.

Locally, each storage server would maintain the mapping between its hashes and the physical disk locations of their corresponding blocks through a local data structure, such as a binary search tree or a local hash table. Since this data structure only contains hashes and pointers, it can typically be held in memory.

Fault Tolerance

CFS [5] achieves fault-tolerance by storing each block not only on the server whose hash immediately succeeds it, but also on a further k successors, where k is the desired degree of redundancy. Since the cryptographic hash function provides a random distribution, successors along the hash space would be well-dispersed across the physical setup of the network, making it statistically unlikely that any local adversity affecting a small subset of the network (such as a power or router fault) would impact the entire sequence of successors. For example, by storing six replicas of each block, it is possible for up to 20% of the servers to fail simultaneously without losing any data [5].

Chord [3] provides stabilization mechanisms for automatically updating the finger tables and successor lists whenever nodes join or depart the network. These are exploited by CFS [5] to dynamically replicate blocks onto new storage servers following node failures, making the network self-repairing.

Load Balancing

The random distribution of the hashes implies that each server is responsible for roughly an equal number of blocks. Since directories are also stored as blocks, they too would get evenly distributed across the network; this circumvents the bottleneck of using centralized servers to provide directory path resolutions [5].

Another convenient consequence of this random distribution is that any client accessing a large number of blocks – possibly even belonging to a single massive file – would be able to issue their requests across all servers concurrently, boosting system throughput.

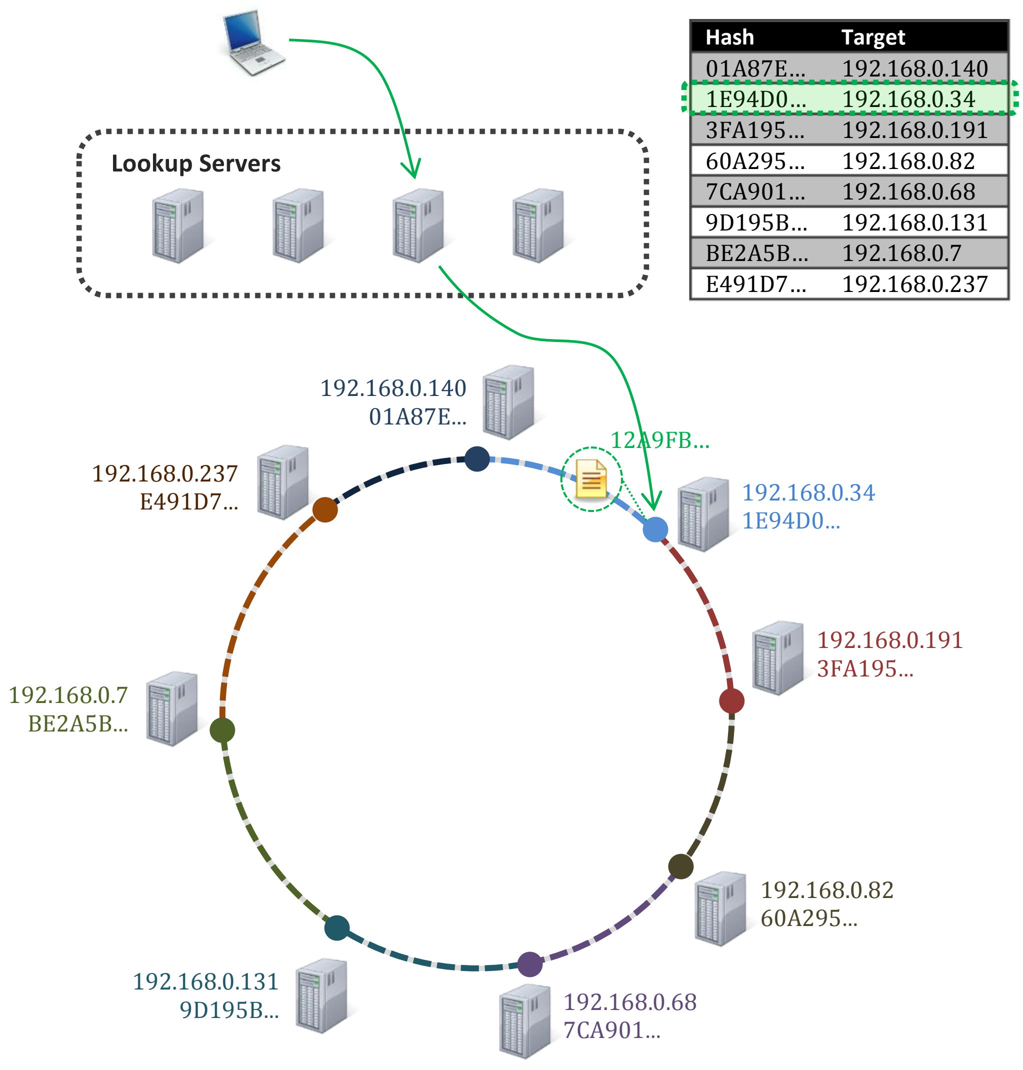
Centralized Lookup

Being designed for peer-to-peer systems (P2P) comprised of transient participants, some characteristics of DHTs tend to be too pessimistic about node failures, having to contend with median lifetimes on the order of hours [15]. For this reason, they sacrifice some efficiency for aggressive robustness – a trade-off that may not be as applicable on Cloud infrastructures, where hardware failures would be significantly less frequent [16]. Furthermore, full decentralization is a core priority in P2P; this is not necessarily desirable on the Cloud, where clients may rely on accessing a core set of trusted servers, and potentially require centrally-administered services such as user authentication and access control [4].

In light of this, we can boost our system's efficiency by introducing an additional layer of lookup servers, whose role is loosely analogous to – albeit much simpler than – the master servers in the Google File System (GFS) [17]. Each lookup server independently maintains a table of all available storage servers, sorted by their hash identifiers. When a client needs to look up the storage server responsible for the block having a particular hash, it may now send the request to a lookup server (picked at random for load balancing), who would forward it directly to the correct storage server. This reduces the path length of any request from a logarithmic number of hops to a constant two-hop scheme [18].

Like master servers in GFS [17], our lookup servers are stateless and operate from memory. Their lookup tables are not persisted, but populated dynamically: When a lookup server starts up, it issues a broadcast requesting the identifier hash of each storage server, and updates its table with all received responses. When a new storage server joins the network, it similarly broadcasts its identifier hash to all lookup servers. Each lookup server may monitor a random subset of the storage servers through heartbeat messages [17]; when a failure is detected, the lookup server should send out a broadcast in order to update the other lookup servers as well as accelerate the stabilization mechanism on the storage servers.

A further improvement that may be considered would be the caching of the lookup table on the client side. This allows clients to bypass the lookup servers and contact the correct storage server directly for most of their requests, thereby reducing the network latency to a single round-trip. Clients may request a callback promise (discussed below) for having lookup servers notify them when the lookup table changes.



Hash	Target
01A87E...	192.168.0.140
1E94D0...	192.168.0.34
3FA195...	192.168.0.191
60A295...	192.168.0.82
7CA901...	192.168.0.68
9D195B...	192.168.0.131
BE2A5B...	192.168.0.7
E491D7...	192.168.0.237

Figure 7. Same lookup as in Figure 6, but using lookup servers to immediately identify the responsible storage server for the requested hash.

Client Access

In order to provide a seamless experience for end-users, including the possibility for disconnected operation, we shall use the same approach as AFS [1] and Coda [2], and provide a client-side module for extending the local filesystem name space. This client module implements two interfaces: one for plugging in to the operating system's virtual filesystem (VFS) on the local workstation [19], and one for communicating with the servers over the network. Within its core, the client module may also implement any additional client-side functionality, such as caching and logging.

By registering itself with the VFS, the client module becomes transparently accessible by user applications (including the shell) through a mount-point on the local name space – any file paths traversing this mount point would get forwarded by the kernel to our module [19]. The API for registering with the VFS may vary across platforms, requiring our module to provide an OS-specific implementation which, among other responsibilities, needs to transform any filesystem structures or metadata received from the storage servers to the format expected by the VFS – Mazières [20] proposes a cross-platform toolkit for improving portability. However, the rest of the functionality of the module, including its caching mechanism and client–server communication protocol, may be implemented in an OS-independent manner, and reused across all targeted OSes.

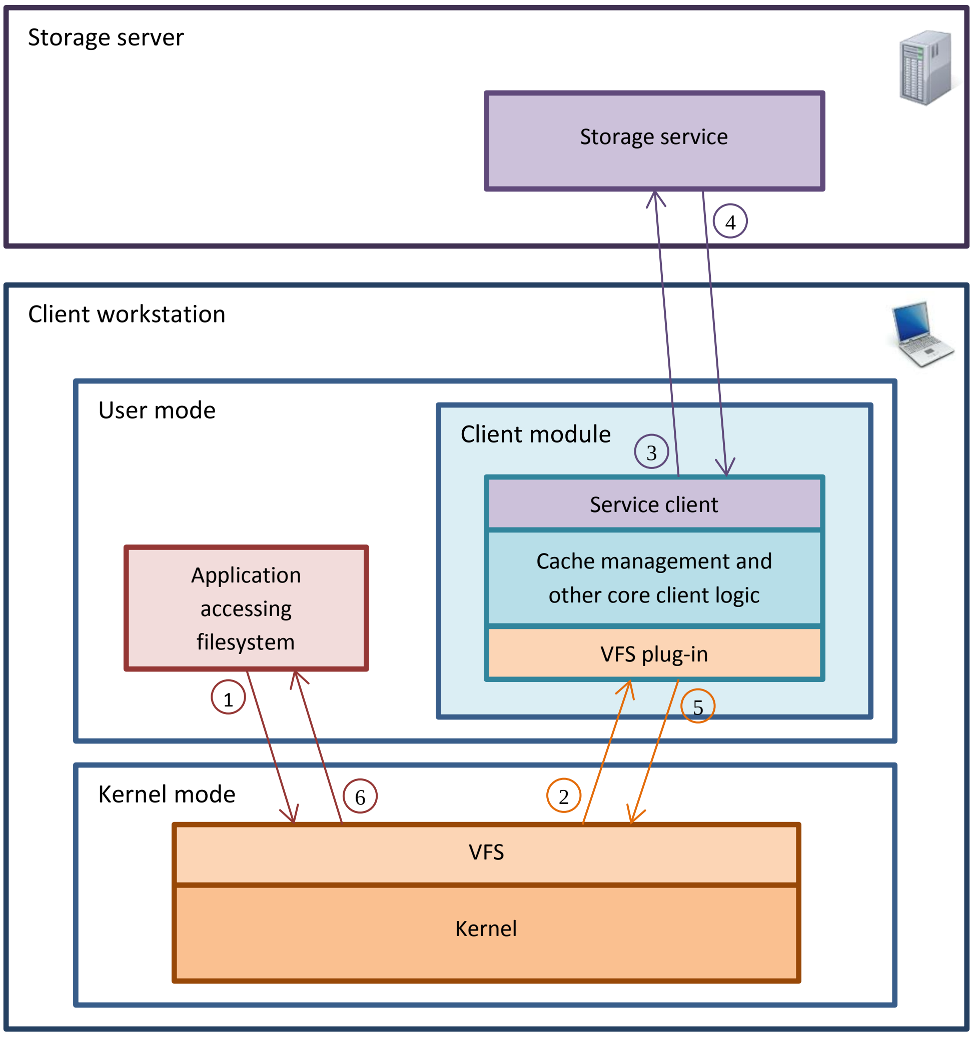


Figure 8. Role of client module for intermediating between the local VFS and the storage servers.

Each client may communicate with the lookup and storage servers using remote procedure calls (RPC). When a file is requested, it should be fetched in entirety and cached within the client module. Since the data blocks for each file would be dispersed across the network, their requests may be issued to – and serviced by – the responsible servers concurrently, thereby boosting throughput. Consequently, for large transfers, the performance of the system would only be bound by the network bandwidth [21], [22], which may substantially outperform disk throughput (particularly for fragmented files) [23]. Additionally, clients should pick which replica to direct each request towards at random, so as to promote load-balancing.

In order to limit network congestion, we will provide write-on-close semantics like in AFS [1]. When a file with saved changes is closed, the client module would upload any data (and pointer) chunks that were affected to their respective storage servers, based on their new hashes. Finally, the client needs to inform the servers responsible for the parent directory to update the hash stored as the file’s entry in order to point to the new data.

Coda [2] specifies that such file updates should be issued by the client to all replicas simultaneously. A change may be considered as successfully committed once it has been confirmed by more than two-thirds of the responsible servers. This allows commits to succeed even in the presence of server failures or network partitions, and ensures, with high probability, that the change would eventually be successfully propagated to all replicas using Byzantine agreement [18], [24].

The immutability of data and pointer blocks guarantees that no write conflicts may arise when several users are simultaneously accessing distinct logical files pointing to the same physical data. When any byte in a resource is changed, its hash would also change, and therefore needs to be stored at a separate address.

However, write conflicts may arise when users simultaneously access the same *logical* file, since updates to the directory block may result in inconsistencies. We shall assume an optimistic concurrency control strategy, where simultaneous updates are permitted locally, but would get detected upon being committed to the servers due to a differing timestamp. We will follow the suggestion of Coda [2] and provide the users with tools to resolve these conflicts manually, similarly to a version-control system.

Finally, we shall also borrow the notion of callback promises from AFS [1] and Coda [2], allowing clients to be notified promptly when a file they have open has been modified by another user. Note that the callback promise needs to be established by the server responsible for the directory block.

Disconnected Operation

Coda [2] caches entire files within its client module in order to improve performance by exploiting both temporal locality (when the same data is accessed again later) and spatial locality (when other portions of the same file are accessed). Kistler & Satyanarayanan [25] argue that this caching may also be used for improving availability. By persisting the cache to the local disk, Coda would ensure that its client module would be capable of permitting access to the files even when the workstation is restarted whilst disconnected.

Whilst AFS [1] uses a least recently used (LRU) algorithm for determining which data to retain in its cache, Coda [2] complements this with “hoard profiles”, through which each user may specify a list of important files that should always be maintained in cache [25].

Any writes that the user performs in disconnected mode would get logged by the client module. Once the client reconnects, it would propagate such updates to the responsible servers, using the same timestamp-based concurrency control as discussed above. It should also re-establish any callback promises for its cached files.

Security

Each client is expected to authenticate with the system before it may access any directories or files. We recommend the use of Kerberos [26], a network authentication protocol that relies on a set of Authentication Servers (AS). Each user would initially supply their password to the client program, which gets hashed using a one-way function and transmitted to the Key Distribution Center (KDC).

The KDC returns a Ticket Granting Ticket (TGT), which may be subsequently used for generating tickets (session keys) whenever the client needs to communicate with any storage server.

Access control only needs to be enforced when accessing directory entries. There is no need to enforce security on the data (or pointer) blocks – since their address constitutes a cryptographic hash, it is computationally infeasible for an attacker to come up with the address for a file to which they did not already have access. Given that the hash space is massive (2^n possibilities, where n is the number of bits in the hash), brute-force enumeration will not yield any results either.

An extrinsic risk is that hashes for files may be leaked to unauthorized users through external applications, since a file's hash in itself is not generally considered to be sensitive information. To avoid this scenario, we should ensure that the hashes used in our system are salted.

Integrity of data and pointer blocks may be verified by re-computing their hashes and checking that it matches the stored one [6]. Per the avalanche effect, corruption in just a single byte would result in a completely different hash. Storage servers may periodically perform such verification in the background, and request the correct data from any of their replicas when corruption is detected.

Harnik et al. [27] demonstrate a side-channel attack that applies to several systems employing cross-user client-side deduplication, whereby an attacker may deduce the prior existence of a specific file based on whether the system recognizes its hash. This attack may be prevented either by moving data deduplication to the server side, or by introducing random delays.

Conclusion

To conclude, we shall summarize how our system design meets all the requirements set out for Onefile.

Single-instance storage of physical files is achieved by deterministically deriving their target address from their contents using a cryptographic hash function. Logical aliases are allowed to span directories and users since they are stored within their respective parent directories, along with their specific filenames and permissions. Consequently, access control is enforced based on this file metadata. The avalanche effect ensures that any change to a file's content would result in a new hash, thereby guaranteeing that the new version is stored at a different address from the old.

Although identical data blocks resolve to a single address, each address is stored not only by the immediate hash successor, but also across a number of replicas. When a server fails, the network uses stabilization mechanisms to ensure that its contents are made available on another server.

Users may work offline by accessing their files from the local cache stored through in the client module. When the connection is re-established, the client module would propagate any logged changes from the cached files to the servers, and only assume success when the servers are in a state that would reach Byzantine agreement, thereby ensuring consistency.

The client module integrates seamlessly with the local operating system's name space by registering itself with the VFS, allowing the kernel to issue callbacks to it whenever the user traverses its mount point.

Finally, we are confident that the core design of our system may scale to infrastructures of any size, since it is modelled on peer-to-peer frameworks that have already successfully demonstrated robust operation on Internet-scale applications.

References

[1] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, ‘Scale and performance in a distributed file system’, *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 51–81, 1988.

[2] M. Satyanarayanan, ‘Scalable, secure, and highly available distributed file access’, *Computer*, vol. 23, no. 5, pp. 9 –18, May 1990.

[3] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, ‘Chord: a scalable peer-to-peer lookup protocol for internet applications’, *Networking, IEEE/ACM Transactions on*, vol. 11, no. 1, pp. 17–32, 2003.

[4] I. Foster and A. Iamnitchi, ‘On death, taxes, and the convergence of peer-to-peer and grid computing’, *Peer-to-Peer Systems II*, pp. 118–128, 2003.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, ‘Wide-area cooperative storage with CFS’, *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 202–215, 2001.

[6] S. Quinlan and S. Dorward, ‘Venti: a new approach to archival storage’, in *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, 2002, vol. 4.

[7] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, ‘A fast file system for UNIX’, *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 3, pp. 181–197, 1984.

[8] J. Li and S. Vuong, ‘An Ontological Framework for Large-Scale Grid Resource Discovery’, in *12th IEEE Symposium on Computers and Communications, 2007. ISCC 2007*, 2007, pp. 757 –762.

[9] B. Zhu, K. Li, and H. Patterson, ‘Avoiding the disk bottleneck in the data domain deduplication file system’, in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2008, pp. 18:1–18:14.

[10] M. Naor and M. Yung, ‘Universal one-way hash functions and their cryptographic applications’, in *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, 1989, pp. 33–43.

[11] R. C. Merkle, ‘A Digital Signature Based on a Conventional Encryption Function’, in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, London, UK, UK, 1988, pp. 369–378.

[12] A. Muthitacharoen, B. Chen, and D. Mazieres, ‘A low-bandwidth network file system’, in *ACM SIGOPS Operating Systems Review*, 2001, vol. 35, pp. 174–187.

[13] D. T. Meyer and W. J. Bolosky, ‘A study of practical deduplication’, *Trans. Storage*, vol. 7, no. 4, pp. 14:1–14:20, Feb. 2012.

[14] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, ‘Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web’, in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, New York, NY, USA, 1997, pp. 654–663.

[15] D. Stutzbach and R. Rejaie, ‘Understanding churn in peer-to-peer networks’, in *Internet Measurement Conference: Proceedings of the 6 th ACM SIGCOMM conference on Internet measurement*, 2006, vol. 25, pp. 189–202.

[16] K. V. Vishwanath and N. Nagappan, ‘Characterizing cloud computing hardware reliability’, in *Proceedings of the 1st ACM symposium on Cloud computing*, New York, NY, USA, 2010, pp. 193–204.

[17] S. Ghemawat, H. Gobioff, and S. T. Leung, ‘The Google file system’, in *ACM SIGOPS Operating Systems Review*, 2003, vol. 37, pp. 29–43.

[18] A. Verma, S. Venkataraman, M. Caesar, and R. H. Campbell, ‘Scalable Storage for Data-Intensive Computing’, *Handbook of Data Intensive Computing*, pp. 109–127, 2011.

[19] S. R. Kleiman and S. Microsystems, ‘Vnodes: An architecture for multiple file system types’, 1986, pp. 238–247.

[20] D. Mazières, ‘A Toolkit for User-Level File Systems’, in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2001, pp. 261–274.

[21] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi, ‘Grid Datafarm Architecture for Petascale Data Intensive Computing’, in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002*, 2002, p. 102.

[22] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, ‘Beowulf: A Parallel Workstation For Scientific Computation’, in *In Proceedings of the 24th International Conference on Parallel Processing*, 1995, pp. 11–14.

[23] J. Zhang, G. Wu, X. Hu, and X. Wu, ‘A Distributed Cache for Hadoop Distributed File System in Real-Time Cloud Services’, in *2012 ACM/IEEE 13th International Conference on Grid Computing (GRID)*, 2012, pp. 12 –21.

[24] L. Lamport, R. Shostak, and M. Pease, ‘The Byzantine generals problem’, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[25] J. J. Kistler and M. Satyanarayanan, ‘Disconnected operation in the Coda File System’, *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 3–25, Feb. 1992.

[26] B. C. Neuman and T. Ts’o, ‘Kerberos: an authentication service for computer networks’, *IEEE Communications Magazine*, vol. 32, no. 9, pp. 33 –38, Sep. 1994.

[27] D. Harnik, B. Pinkas, and A. Shulman-Peleg, ‘Side Channels in Cloud Services: Deduplication in Cloud Storage’, *IEEE Security Privacy*, vol. 8, no. 6, pp. 40 –47, Dec. 2010.