vanDisk: An Exploration in Peer-To-Peer Collaborative Back-up Storage

Amir Javidan, Tony Angerilli, Armin Barhashary, Guy Lemieux, Roman Lisagor, Matei Ripeanu
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC
Contact Authors: amir.javidan@gmail.com, or {lemieux | ripeanu}@ece.ubc.ca

***Abstract*—As personal computers become an integral part of our daily lives, huge volumes of data need to be reliably managed and archived. Uncorrelated failures within a set of independent personal computers offer the promise of low-cost, reliable data storage. The vanDisk project attempts to realize this promise.**

The main assumption of our project is that users are willing to donate raw storage space to their peers to increase the reliability of their own data. In our system, users offer a portion of their disks to be used as backup space for other users in exchange for space to store backup copies of their own data, thus decreasing the possibility of catastrophic data loss.

A number of characteristics differentiate vanDisk from existing projects that explore this space. First, unlike existing projects that that increase redundancy at the data-block or file level, vanDisk operates at the disk level. This substantially simplifies data management and reduces management overhead at the cost of marginally higher recovery costs from partial failure. Second, all data-related operations are transparently replicated at the data source. Third, our design includes an orthogonal component to manage space and bandwidth.

Our system is integrated with Microsoft Windows and offers users a virtual drive that transparently replicates data across multiple machines. As well, a complete, original copy of the data is always available on the user's own system. We have modified TrueCrypt, an open source virtual disk package that offers data confidentiality through encryption, and we have added a new driver layer that redirects and replicates all IO requests to a set of Network Block Device servers offered by the peers to store replicated data. Additionally, we use simple data encoding to offer user-tunable tradeoffs between space overheads, compute overheads, and data reliability.

Keyword: peer-to-peer data backup, virtual drive, collaborative data backup.

I. Introduction

Computer usage has become an integral part of our daily lives. As pervasive computing increasingly leads to 'everyware' technology, reliable safe-keeping of data has become a significant issue. Part of the problem stems from neglect to proactively backup valuable data, often a result of lack of a transparent, automated backup technology. Another aspect of the problem is that popular storage media such as CDs, DVDs, or additional hard drives often fail to provide an adequate level of durability.

Our project aims to alleviate these long-term data backup problems by introducing the notion of a virtual array of network disks, or *vanDisk*, which transparently replicates user data over disks on multiple remote machines to increase data reliability. In doing so, the system eliminates the need for proactive action from the user to backup data, while drastically improving the reliability of stored data by continuously monitoring the availability of the machines on which data is replicated. We have prototyped vanDisk thus far as a sequence of 4th year undergraduate projects to investigate the main tradeoffs, its feasibility and its practicality.

The main assumption underlying vanDisk is that users are willing to donate raw storage space to increase the reliability of their data. Thus, users offer a portion of their disks to be used as backup for other users' data, in exchange for space to backup their own data. Characterizations of desktop disk space usage in both corporate [1] and academic [2] environments show that this assumption is realistic: most desktops have at least half of their disk space unused.

Three characteristics differentiate vanDisk from other projects that explore the peer-to-peer data storage space [1-6]. First, unlike existing projects that provide increased data availability or durability through increased redundancy at the data-block or file level, vanDisk operates at the (virtual) disk level. In brief, this choice offers reduced management overhead at the cost of slightly larger recovery times from partial failures. Usually, management overhead is proportional to the number of objects the system has to provide durability for by tracking their replication levels. vanDisk works at coarse granularity, the virtual disk, and thus has lower management overheads than systems that use block-level replication. The coarse granularity has an impact on failure recovery characteristics: if virtual disks need to be locked for writes during the recovery process, the impact of a failure on a single client is difficult to mask. However, as the main goal of vanDisk is to provide data durability rather than availability [7], rapid recovery from failures is a secondary concern.

A second characteristic that differentiates vanDisk is that all data-related operations are transparently replicated at the data source: all disk operations are captured by a vanDisk device driver. While read operations can be served by any available replica, write operations are spread over the entire set of replicas. Optimistic consistency protocols ensure that writes complete in spite of storage node failures. Finally, our design includes an orthogonal component that manages storage space and bandwidth contributions of a system to discourage freeloading.

The rest of this paper is organized as follows. The next section offers a vanDisk overview and a discussion of the main design choices. Section III discusses the data coding technologies used to control the tradeoff between space overheads, compute overheads, and data reliability. Section IV presents vanDisk prototype implementation and Section V summarizes our experience.

II. System overview

VanDisk facilitates the seamless, durable storage for a *client node* by providing a means to mount a logical volume, a virtual drive, which appears to function like a local hard disk. Data operations on the virtual drive are actually delegated to the remote *storage nodes*. Note that each node that participates in the system concurrently plays both roles of client and storage node: as a client the node backs its data up on other nodes, while as a storage node it accepts to store data on behalf of remote nodes.

In our target environment, desktop nodes connected by a local area network, individual nodes are intermittently available and existing nodes might fail and leave the system for good. As a result, the important tradeoff to consider is between the availability and durability of the data and the redundancy involved in storing it [7, 8]. We use erasure encoding [9] to offer tunable tradeoffs between space overheads, compute overheads, and data availability and durability.

A. Design Considerations

The peer-to-peer nature of the system implies that it must be capable to tolerate the following events:

- *Graceful departure:* a storage node may announce its intention to go 'offline' and be temporarily unavailable for storage.
- *Unanounced temporary failure:* a storage node may suddenly fail, due to a network interruption or a crash. The departure is not announced to the system.
- *Permanent failure/departure:* a node may permanently leave the system, possibly as a result of a failed hard disk or of discarding of the computer. This scenario is equivalent to either of the above two, except that the node does not return to an available state.
- *Node arrival*: a node may come back 'online', once again make its stored data and free disk space available.
- *Node join:* a new node may join the current set of nodes.

When a storage node *P* is detected as unavailable, the reliability of the data stored in the system is reduced and the system must attempt to restore it. This is done by using available nodes to reconstruct the data *P* was storing, and then making that data available for client operations. However, attention must be paid to consistency issues with regards to write operations during restoration process. Furthermore, the system must decide where to store *P's* reconstructed data.

In the opposite scenario, when a storage node *P* is detected as once again available after being down for a period of time, the system can either update *P* with the newest copy of the data it is responsible for storing, or consider *P* as an unused node, and continue using the existing substitute node

B. Design Choices

To detect node status the system uses a Discovery Service. The Discovery Service expects all storage nodes to send a regular *keepalive* message. If the Discovery Service detects that a node failed to send keepalive messages for a predefined number of rounds it infers that the node has left the system. The Discovery Service continues to store the state of the node. If the node does not come back online within a predefined period of time the Discovery Service considers the node permanently failed and deletes all state related to that node.

This design has the advantage that nodes can naturally join the system just by starting to send their keepalive messages. Additionally, the Discovery Service does not initiate any communication but merely waits for keepalive messages.

Once the Discovery Service detects a change in a storage node status, it sends this information to the node that has mounted the virtual disk, the 'client' node in our terminology. This way, the client is able to depend strictly on the Discovery Service to make policy decisions as to how long to wait before declaring a storage node unavailable or permanently failed.

After a node *P* has been declared unavailable for a certain amount of time, the Discovery Service asks the Reconstruction Service to reconstruct *P's* data and make it available for the client to utilize. The Reconstruction Service will then read the dataset stored at a sufficient number of nodes, in order to reconstruct the full data stored at *P*. During this process, it also synchronizes with the client, to ensure that write operations are queued until the reconstruction process completes. Once the reconstruction process completes, the Reconstruction Service asks the Space Management Service to make available storage space to store node *P*'s data. Ideally, the Space Management Service is able to select from a pool of unused storage nodes to store node *P*'s data. The Space Management Service then notifies the Reconstruction Service about the details of how to access the newly available storage space. The Reconstruction Service then stores node *P*'s reconstructed data at the newly available storage space, and notifies the Discovery Service as to how to access the reconstructed data. The Discovery Service then notifies the client of the new replica information, and allows the client to resume any pending write operations. Since the reconstruction process is an expensive operation, this scheme attempts to make it transparent to the user while minimizing its performance impact.

Since nodes may experience brief network interruptions, the client node, attempts to perform read/write operations at each of the storage nodes, even when facing failures (failed reads are retried at different replicas while failed writes are buffered for delayed retries). It is only when the Discovery Service notifies the client node of a change in the replication set that the client node considers a storage node to be unavailable. This design provides a clear separation of concerns between the Discovery Service in charge with detecting node failures and client nodes in charge with individual read/write operations.

C. Discussion

The design used to control replication is simple and our experience has proven it effective. While, the current vanDisk implementation uses centralized implementations for the services mentioned above, our design does not preclude a decentralized implementation. In a decentralized scheme, the storage nodes cooperate with each other to detect changes in availability of all storage nodes, reconstruct a failed nodes data, find new storage space for the newly reconstructed data, and communicate replica location changes to the client.

In order to provide these services, each group of storage nodes responsible for a certain virtual disk uses a leader-election algorithm to identify a particular node as responsible for group membership management, detecting membership changes, and failure recovery. Ideally, leader selection takes into account each node availability patterns and proximity to the client. If the leader fails, the remaining storage nodes elect a new leader, and ensure that the management responsibilities are seamlessly ported to the new leader. The drawback of a distributed management scheme is the increased overhead involved with the storage nodes having to communicate and agree with one another. Additionally storage nodes maintain the metadata to track and maintain the systems' state.

III. Erasure Codes for Data Distribution

Erasure codes provide a means to transform a set of k data blocks into a set of $n > k$ blocks, such that the original set of k data-blocks can be recovered from a subset of the encoded n blocks [10, 11]. The advantage of using erasure codes is that the system can provide high availability while using less disk space than pure replication. However, erasure codes also introduce computational overheads for encoding and decoding. This section compares various coding techniques [11].

Reed-Solomon (RS) codes [12] are a key component of CD and DVD technology and have been used extensively in data communications. RS codes can detect any $m=n\text{-}k$ errors in a set of n total encoded blocks, and can correct up to $m/2$ failures. In most cases, RS codes perform well when the value of n is small, but have been recently superseded by more computationally-efficient encoding methods such as Tornado Codes [13].

Low-Density Parity-Check (LDPC) codes are more computationally-efficient than RS codes and among the most effective known codes. LDPC codes are *systematic:* the n resulting blocks include the k source blocks verbatim and an additional $m=n\text{-}k$ new parity blocks [12]. This property allows for faster encoding and decoding times.

Fountain codes [14] encode a block of data into an infinite sequence of parity blocks. As a result, there is no requirement for a predefined value for n. In our system, this is a valuable property since it allows adding new replicas without having to re-encode the full dataset. However, fountain codes are patent-protected and no open source implementation is available [15].

RS and LDPC codes are the most promising for our software implementation. The most important characteristics are their encoding/decoding time and the number of blocks required to reconstruct a valid data set, commonly referred to as the Maximum Distance Separable (MDS). When the MDS is equal to k, the code is considered *optimal*. RS codes are optimal for arbitrary values of n and m, and thus can guarantee that any m erasures are always tolerated. However, this flexibility and reliability is provided at the cost of computational overhead. In addition, RS decoding requires k blocks before decoding can begin, whereas LDPC decoding can take place on-the-fly. This limitation can make RS codes prohibitively expensive for large values of n. In contrast, LDPC codes are sub-optimal, however their space overheads are low and they provide much faster encoding and decoding times. We use LDPC codes for vanDisk.

IV. Prototype Implementation

A. The Client Node

To implement the virtual disk abstraction, we started with TrueCrypt [16], an open-source software that allows mounting an encrypted virtual drive from a either file or raw disk volume (aka partition). All I/O requests received by the virtual file system are captured by the TrueCrypt driver, which then performs encryption/decryption to the data to be written to (respectively read from) the disk. Then, the driver either applies I/O operations to the file, or forwards the request to the volume driver if it is backed by a raw volume. To divert these calls, we inserted a kernel-mode driver into this chain which forwards requests from the TrueCrypt driver to our user-level process called RAIDClient.

The RAIDClient performs all reliability-related erasure encoding or decoding and contacts the data storage nodes to retrieve or store the cyphertext data. By delegating kernel-level IO requests to a user-level process, the amount of kernel-level code is reduced. This reduces the I/O performance, particularly due to context switching, but it is easier to code and debug.

Using a disk-level abstraction, as opposed to a file system abstraction, drastically reduces the accounting and metadata management overhead and offers better transparency. However, we must sacrifice some performance involved with recovering from a node failure as outlined in section II.

B. *The Storage Node*

To implement the storage nodes we use the Network Block Device (NBD) [17] open source library. NBD allows the use of a file as a block device, and provides its own protocol for data transfer between a client and a server. By using files as the storage mechanism on the data storage nodes rather than separate partitions or physical disks, storage nodes can easily export space for consumption by one or more clients without the complications of re-partitioning disk drives.

When a read operation from a particular storage node fails, the client can reconstruct the requested data as long as roughly k of the n storage nodes are available (this depends in part on the additional constraints the symmetric LDPC codes bring). Write operations generate several further complications compared to reads. If a write to a particular storage node fails, then the client buffers the write operation. The next time it attempts to read or write to that node, assuming the node becomes available once again, it will first apply the queued writes before any other operation to ensure consistency. However, once the Discover Service declares a storage node failed and notifies the client, this will begin reconstructing the missing data on a new storage node. The client can then discard all queued operations related to the failed storage node.

C. *Feasibility Experiment*

To test the functionality of our prototype we have deployed and tested vanDisk in a minimal setting. We have used seven desktop computers as follows: one client node, five storage nodes, and one Discovery Service node.

The system was configured to tolerate up to two out of the four of the storage nodes failures. An additional storage node was left unused (spare) to store reconstructed data and replace failed nodes on the fly. The server nodes were relatively old machines (500MHz Intel Pentium III, 128MB). A virtual drive was mounted on the client machine, and a 30 MB video file was stored on the drive. Node reconstruction and fail-over behavior was observed as the video file was viewed on the client machine. The video file played smoothly without any interruption through both one and two storage node failures, and experienced no significant performance losses during the reconstruction process. The encoding and decoding overheads were negligible as compared to the communication overheads. The client CPU consumption remained below 10% during intense read and write operations, and CPU usage was negligible for the storage nodes. While the system was implemented as a proof-of-concept rather than optimized for performance, the system demonstrated reasonable behavior under stress-test conditions.

V. Summary

The vanDisk project aims to alleviate long-term data backup problems by introducing the notion of a virtual array of network disks that transparently replicates a user's data over multiple remote machines to increase data availability and durability. vanDisk, eliminates the need for proactive action from the user to backup data while drastically improving the durability of stored data by continuously monitoring the remote machines on which data is replicated.

We have prototyped vanDisk as a sequence of 4th year undergraduate projects. Even though our vanDisk prototype is still early stages, *e.g.*, it is not optimized for performance and uses a number of centralized components, is proves the feasibility and practicality of our virtual disk based approach.

VI. References

[1] W. J. Bolosky, J. R. Douceur, D. Ely, et al., "Title Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs," International Conference on Measurement and Modeling of Computer Systems(SIGMETRICS), 2000.

[2] S. S. Vazhkudai, X. Ma, V. W. Freeh, et al., "Constructing collaborative desktop storage caches for large scientific datasets," vol. 2, pp. 221 - 254, 2006.

[3] R. Bhagwan, K. Tati, Y. Cheng, et al., "TotalRecall: System Support for Automated Availability Management," NSDI'04, 2004.

[4] J. Kubiatowicz, D. Bindel, Y. Chen, et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), Cambridge, MA, 2000.

[5] M. Lillibridge, S. Elnikety, A. Birrell, et al., "A Cooperative Internet Backup Scheme," USENIX'03, San Antonio, TX, 2003.

[6] M. Landers, H. Zhang, and K.-L. Tan, "PeerStore: Better Performance by Relaxing in Peer-to-Peer Backup," 4th IEEE International Conference on Peer-to-Peer Computing, 2004.

[7] B.-G. Chun, F. Dabek, A. Haeberlen, et al., "Efficient Replica Maintenance for Distributed Storage Systems," 3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI), San Jose, CA, 2006.

[8] C. Gkantsidis and P. R. Rodriguez, "Network coding for large scale content distribution," 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005), Miami, FL, 2005.

[9] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, pp. 24-36, 1997.

[10] A. G. Dimakis, P. B. Godfrey, M. Wainwright, et al., "Network Coding for Peer-to-Peer Storage," Infocom, 2007.

[11] R. L. Collins and J. S. Plank, "Assessing the Performance of Erasure Codes in the Wide-Area," International Conference on Dependable Systems and Networks (DSN), Yokohama, Japan, 2005.

[12] T. K. Moon, *Error Correction Coding, Mathematical Methods and Algorithms*: Wiley, 2005.

[13] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, et al., "Practical loss-resilient codes," 29th ACM Symposium on Theory of Computing, El Paso, TX, 1998.

[14] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, et al., "Improved Low-Density Parity-Check Codes Using Irregular Graphs and Belief Propagation," IEEE International Symposium on Information Theory, 1998.

[15] C. Neumann and V. Roca, "Design, Evaluation and Comparison of Four Large Block FEC Codecs, LDPC, LDGM, LDGM Staircase and LDGM Triangle, plus a Reed-Solomon Small Block FEC Codec," INRIA Rhone-Alpres, Planete Research Team, Lyon, France 2004.

[16] TrueCrypt: Free open-source disk encryption software; http://www.truecrypt.org/, 2007, accessed on: April 2006

[17] P. T. Ares, "The Network Block Device," *The Linux Journal*, 2000.