PeerSafe

Brian C. Howard

*with implementation help from*
Joshua Barnard

December, 2003

I. Abstract:

In an age when computer manufacturers are equipping their PCs with hard drives of increasing capacity and decreasing quality, the data people store on them is in peril. With the widespread use of the personal computer, the data which people most value has moved into the digital realm. Despite knowing that the risk of hard drive failure is present, most people do little or nothing to back up their data, since most current options are either expensive or labor intensive.

PeerSafe addresses this problem by securely distributing a user’s data over a peer-to-peer network, using otherwise unused space on a user’s drive to store chunks of data from other users of the network, thus guaranteeing data security for everyone on the network. By being decentralized and free, PeerSafe has economic advantages over expensive proprietary centralized backup services, as well as the advantage of not being as vulnerable to wide-area disasters (hurricanes, fires, etc) which might wipe out a single centralized facility. Through use of cryptography and a clever division of data, PeerSafe guards the privacy of data entrusted to it from prying eyes. PeerSafe is designed to take advantage of already present resources to safeguard the increasing volume of personal digital information.

II. Introduction:

PeerSafe aims to be both distributed and decentralized, meaning that each node in the system is fundamentally like every other node, and that no single node directs or guides the system. By being distributed and decentralized, PeerSafe avoids the single-point-of-failure problems with centralized systems.

In order to accomplish this, PeerSafe relies on an overlayed distributed hash table (DHT) to deterministically map abstract addresses specific to PeerSafe to actual internet addresses without relying on a strict hierarchy as in systems like DNS. Use of a DHT allows for data to be quickly and easily located when it is needed for reconstruction of a user’s data.
III. Theoretical System Details

When PeerSafe first starts, it generates an empty disk image to hold the data which the user would like to have backed up. A second disk image, which is larger by a certain factor than the first image is also created. This image is used for the data belonging to other nodes which this node will hold for them. The factor by which the sizes of the images differ will henceforth be called the ratio.

The ratio determines the level of redundancy the system tolerates, as well as the time allowed between a drive failure and recovery. The greater the ratio, the more redundant copies of the same data may reside in the system. A greater ratio also means that more free space will be available in the system as a whole to hold the data of a crashed user, since that user is essentially free-loading – keeping his data in the network, but hosting none in exchange. The precise optimal value for the ratio depends very much on these and other factors, and is beyond the scope of this paper (see the Further Work section).

After the images are set up, the system generates a long series of random bytes which will serve as its system identifier. The length of the array (24 bytes in the reference implementation) should be sufficiently large to make it incredibly improbable that any two nodes would generate identical identifiers. This identifier is used later in conjunction with the DHT.

After the creation of the system identifier, the user is presented with the opportunity to add and remove files from the images. When the user is satisfied, they tell the node to publish their data to the system. The node then dynamically generates chunks, or blocks of data of a standardized size, from the image.

The user's data image is constrained in its size to being divisible into sets each of which are \( n \) times the size of a single chunk. Using the paradigm borrowed from RAID level 5\(^1\), the node constructs \( n+1 \) chunks from the data in each set. So for \( n = 4 \) and a chunk's size = 1, possible image sizes would be 4, 8, 12, etc. An image of size 12 would be split into 3 sets of 4 parts each. Each set would produce 5 chunks (the 4 parts, and 1 part parity). Excepting the parity chunk, the data contained in chunk \( i \) represents every \( n \)th byte of the data from the set, starting with byte \( i \). The parity chunk then contains parity data for the other \( n \) chunks – that is, byte \( p \) in the parity chunk is formed by XORing the corresponding bytes in the other \( n \) chunks.

This method of dividing the data has several advantages over a simple partitioning. Foremost, it is possible to reconstruct the data in a set with only \( n \) of the \( n+1 \) chunks. Achieving the same effect without the use of parity data would require double the amount of data as the original set; this method uses only \( n+1/n \) times the

---

space. In addition, each chunk is published twice, representing a space usage of $2(n+1/n)$ of the original data with the ability to sustain the loss of any 3 chunks without loss of data. A second advantage is the inherent scrambling of the original data. Should the encryption be cracked, it is much harder to gain usable information when the cracked file contains only every nth byte of the original.

As the chunk is generated, it is also encrypted using a block cipher to guarantee security. Any block cipher with a sufficiently large block works (the reference implementation uses blowfish\textsuperscript{2} in ECB mode with a key size of 128).

So the node dynamically generates each chunk (as identified by set, part and copy) from the disk image, encrypting it as it attempts to send it into the system. A chunk identifier address is deterministically generated by hashing a combination of set, part, copy and the system identifier from earlier. This chunk identifier is of the same format and length as the system identifier. The hash function is a one-way hash, such that given set, part, copy, and system identifier, the chunk identifier may be calculated, but none of the other fields may be calculated given only the chunk identifier. It must also have the property of producing seemingly random hash values (i.e. values that are evenly distributed over the space). An algorithm like SHA1\textsuperscript{3} or MD5\textsuperscript{4} may be used to accomplish this.

The DHT is then used to map the chunk identifier to another node. It is important to note that this node is effectively chosen at random, since the system id of the originating node is random, and the chunk identifier is hashed from this in part. A communication follows between the original node and the contacted node. If the contacted node already has a chunk from the original node, it will deny the request to take this chunk. If the contacted node has a chunk such that the originator does not already possess a chunk from that node, then that chunk is exchanged for the chunk being published by the originator. Failing that, if the contacted node has space, it will magnanimously accept the chunk without asking that the originator take one in exchange.

After the actual data of the chunk is transferred, a series of DHT addresses based on the identifier of the chunk are deterministically created (in the reference implementation, by negating and reordering the bytes of the chunk id). The nodes in the DHT responsible for these addresses are designated as watchers for this chunk, and are sent message informing them that the chunk is now being held by the contacted host (the holder).

By having watchers in the DHT who keep track of the actual location of the chunk, the overhead of transferring the chunk over the internet whenever its DHT address

\textsuperscript{2} Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish), by Bruce Schneier, 1994.

\textsuperscript{3} RFC3174: US Secure Hash Algorithm (SHA1), by D. Eastlake, 3\textsuperscript{rd}, 2001.

hashes to a different node is avoided. In active systems, this layer of abstraction provides significant savings.

The above procedure is repeated for each chunk. At this point, the system ceases action with the exception of 3 timed tasks. The first task is to locate the holder of each chunk and contact them, confirming that they still have the chunk. An arbitrary byte from that chunk could be requested for proof that the holder still has the data. If the chunk has been exchanged, the watchers would have been updated, and the new holder would be contacted. If any one of the watchers were down when the transfer occurred, they would not respond to the request to locate the chunk, or would be out voted by the other, up to date watchers. At this point the holder updates a timestamp on the chunk, indicating that its owner is alive in the system. Any watchers with out of date information are also updated. The purpose of this is to eliminate dead data in the system and to make room for regeneration of current data.

Accordingly, the second task iterates through the chunks held by a node, purging those which have not been checked up on in a long time. If a node suffers a failure and drops from the network, the expiration time of the chunks determines how long the user has to recover his data from another computer. The longer the expiration time, the longer the user may freeload by keeping his data in the system. Eventually though, a user who does not recover will have his data be purged to make room for the data of productive users.

The third task deals with the loss of stored chunks associated with a node going down. When a node is persistently unable to contact the holder of a chunk, it republishes that chunk into the system following the above procedure. The watchers are updated to watch the new location. If the old holder were to miraculously return after a prolonged absence, the data he held would no longer be pointed to by the watchers, and thus never checked up on, and so eventually purged.

Operation continues indefinitely in this mode, except when a user alters his image and chooses to republish it. In that scenario, chunks from all affected sets of the image would be republished to their holders in the DHT.

Successful operation of the system is contingent upon the user retaining three pieces of data, each very small. First the user must keep his system identifier, since without it, his node would be unable to find the watchers which monitor the location of his chunks. Second, he must keep the cryptographic key which was used to encrypt his data. Without this, his recovered data would be unintelligible. Lastly, he must keep the number of sets in his image, so that he can request all of the chunks in the image (it is assumed that the number of parts (n+1) and copies (2) are system constants). This small amount of data could easily be written to a floppy, printed out, or tattooed onto the user for safe keeping.

In the event of a failure, PeerSafe is started up in recovery mode, and given the three pieces of data above. It then proceeds to randomly request chunks by contacting the
host the watchers report as the holder. Once the chunk has been transferred, the holder is instructed to purge the chunk, and the watchers their record of the chunk’s location. When the node has acquired sufficient chunks with which to reconstruct the image, it does so, and extracts all the files contained therein, thus restoring the user’s precious data.

Scalability of the system is largely dependent upon the scalability of the underlying DHT, since the PeerSafe system itself requires very little state be kept that is related to the size of the network. Many of the factors in the system are tunable to trade redundancy for resource consumption – the ratio, the number of copies, the number of watchers, etc. Since movement of the chunks is kept to a minimum, nodes joining the network cause little disruption to data already in place, and changes in the DHT do not cause any movement of chunks (only locations are transferred).

IV. Reference Implementation and Analysis

To demonstrate the soundness of the concept, a reference implementation of PeerSafe was written. The implementation is in Java and used off-the-shelf packages for encryption, while custom classes were written for the DHT, disk images and parity splitting.

The DHT underlying the system was written to model Content Addressable Networks (CAN)\(^5\), although Tapestry\(^6\) was considered as a model as well. In CAN the space of the network can be visualized as a Cartesian space, where any point can be identified by a set of coordinates corresponding to the dimensions. For simplicity, the reference implementation uses a two dimensional space in the shape of a square (rather than wrapping it around into a torus, as in the paper). The byte arrays which represent system and chunk identifiers are then split in half, where the first half is used as the X coordinate and the second half as the Y coordinate. Routing is handled much as in the original, with nodes passing messages in the general direction of the address of the destination. For further explanation, see the CAN paper.

For disk images, a series of two classes were written: one such that the size given to it represented the size of the image file as on the physical drive, the second such that the size represented the free space contained in the image. This was done so that the first type could be split evenly into sets and parts, while the second could be used to store chunks without wasted space. Access to the images is provided through a console where users can add, remove, and update files. The images do not have any advanced consistency features, as that was not the focus of the project.


The encryption used was through the JCE (Java Cryptographic Extension) framework, specifically through the blowfish algorithm which comes packaged with JCE. By using off the shelf cryptography reliability was achieved and time spent on redundant work was reduced.

The division of the images into chunks and the recombination of chunks back into images was also custom coded for the project since no suitable code could be found in the public domain.

The reference implementation performed much as expecting, proving the validity of the concept. Although no vigorous testing of its scalability was done, since scalability is far more dependent on the underlying DHT than on the algorithms comprising PeerSafe, it is safe to assume that coupled with the correct DHT, PeerSafe would scale well.

The fault-tolerant properties of the system (again, apart from those of the DHT) also performed as expected, indicating that, with more robust disk image packages, PeerSafe has the potential to fulfill its purpose.

V. Further Work

There are several areas of PeerSafe which could be further investigated to the benefit of the accident-prone everywhere.

First and foremost, a mathematical analysis of the relationship between the ratio, the expiration time, and the average failure rate of hard drives is badly needed to determine the optimal ratio. One idea for extension was to allow users to choose the ratio instead of having it as a system constant. In return for a larger ratio (agreeing to store more of other users' data for the same quantity of yours), expiration time for a node's data could be extended proportionately. While this sounds like a good idea initially, when such a node went down, the system would be doubly penalized, since a large quantity of stored data would have to be regenerated into the system, and that user's data would be freeloading for longer. Mathematical resources equal to the task of analyzing this interaction are above those of the author.

Secondly, many parts of the reference implementation could be improved. Stronger encryption, more robust disk images and a more efficient multiple reality CAN would all contribute significantly to making it possible to realistically assess PeerSafe's suitability for public use.

Lastly, while probabilistically random distribution of chunks is certainly not a bad thing, and was deliberately chosen to keep the system simple, more advanced methods of distributing chunks could be used. For example, nodes which had high-speed connections to each other might prefer exchanging chunks with each other. Perhaps, the two copies of the data might be deliberately pushed towards nodes in different
geographic areas to reduce the effects of wide area disaster, or nodes might be rated with a reliability index which could allow the system to intelligently store multiple chunks from a single host. Any of these possibilities would make the system greatly more complex. While they were all considered (and would all be interesting to explore), simplicity was one of the goals.