# Transaction-based Charging in Mnemosyne: a Peer-to-Peer Steganographic Storage System

Timothy Roscoe[1] and Steven Hand[2]

[1] Sprint Advanced Technology Laboratory, Burlingame, CA 94010, USA
[2] University of Cambridge Computer Laboratory, Cambridge, CB3 0FD, UK

**Abstract.** Mnemosyne is a peer-to-peer steganographic storage system: one in which the existence of a user's files cannot be verified without a key. This paper applies the techniques used in Mnemosyne—erasure codes and anonymous block writing—to move most of the administrative overhead of a commercial storage service over to the client, resulting in cost savings for the service provider.
The contribution of this paper is to present a radically alternative way of charging for storage services. In place of renting some amount of space for some period of time, systems like Mnemosyne allow more flexible billing models closer to those proposed for network bandwidth, including versions of congestion pricing. We show how a reliable, commercial storage service using is feasible, and examine the details of the tradeoff it offers compared with conventional storage services.

## 1 Introduction

This paper describes a novel paradigm for a distributed storage service built over a peer-to-peer network. The benefits of the approach are extreme simplicity of operation with compared with traditional storage services.

Several current research efforts are building Internet-scale object storage systems by addressing the problem of distributing the functionality of an object store over a peer-to-peer network. This results in self-organising distributed object storage that provides high availability in the face of node failure or network partition. However, such systems share with traditional, more centralised storage systems the complexity that comes from keeping track of large numbers of users and multiplexing storage space among them. Billing for storage adds even more complexity (and administrative overhead) to the system.

The original goal for Mnemosyne[3], described in [1], was to provide extremely high levels of privacy for low-volume, high-value data. As part of this, instead of maintaining space allocation at the server, Mnemosyne holds *no* information at the servers as to which blocks are in use or by whom, and instead relies solely on erasure codes to prevent each user's data being destroyed by the others' activities.

[3] Pronounced *ne moz'nē*.

In this paper, we investigate the feasibility of this approach applied to general distributed storage. In particular we are interested in the costs and benefits of the Mnemosyne approach over traditional storage service models. We show that Mnemosyne in combination with charging for write transactions, rather than storage space *per se*, moves most of the complexity of storage service into the client software, resulting in extreme simplicity (and therefore low administrative overhead) for service providers.

The rest of this paper is structured as follows. In section 2 we briefly review the peer-to-peer application space and position Mnemosyne in this space, both as a highly secure system and as the commercial storage and messaging service we describe here. This latter application of Mnemosyne sidesteps a number of serious design challenges that have recently come to light with peer-to-peer systems.

In section 3 we give a functional overview of Mnemosyne, including a brief description of the current implementation.

In section 4 we use simulation to investigate what kinds of integrity guarantees Mnemosyne is capable of delivering, and what this integrity costs in extra disk storage over conventional systems. In particular, we derive a measure of the "effective capacity" of a Mnemosyne system, which we use to motivate the next section.

Section 5 explores the implications of the findings in section 4 for running a commercial peer-to-peer storage service based on Mnemosyne. We give several different models of charging, and show how what calculations clients must make to use the system, and how service providers decide how much to provision their system for a given effective capacity.

Finally, section 6 concludes with a summary of the tradeoff that Mnemosyne offers.

## 2 Context & Related Work

Perhaps the defining characteristic of peer-to-peer systems is their ability to self-organize — new nodes can join and leave the network without disruption or the need for central coordination or control.

This self-organisation and absence of central control has been exploited to produce systems with strong properties of anonymity and resistance to censorship, such as Freenet [2], FreeHaven [3], and Publius [4].

Research peer-to-peer systems have also appeared which address the scalability problems with early file sharing networks like Gnutella [5]. Projects like Tapestry [6], CAN [7], Chord [8] and Pastry [9] aim to provide robust and highly available generic *distributed hash table* (DHT) functionality; that is, they logically provide an operation `lookup(key)` which maps from an opaque bit string to a node address. In practice, most implementations provide "route message to key" functionality which can then be used to build a variety of applications, for example Oceanstore [10], Bayeux [11], CFS [12], PAST [13], and SCRIBE [14].

The original motivation for Mnemosyne involved a combination of privacy and storage service. Mnemosyne is a distributed *steganographic* file system. A steganographic file system [15] has the property that it gives a user strong protection against being compelled to disclose (all) its contents. Whereas in a cryptographic file system, attackers not in possession of the secret are unable to acquire the contents of files, in a steganographic system they cannot even gain information about whether a given file is present or not. Mnemosyne achieves this property by spreading data pseudo-randomly throughout a peer-to-peer network.

Recently, a combination of deployment experience and research has pointed out a number of vulnerabilities of peer-to-peer systems, both the adversarial attacks and pathological (but common) traffic and usage patterns. For example:

- Load Skew: most P2P schemes assume all participants are equal (i.e. peers) yet studies have shown that node capabilities and user behaviour vary greatly [16, 17]. This negates the basic design assumptions and requires new, non-uniform solutions, such as supernodes in Gnutella [18].
- Untrustworthy Peers: since anyone can join at any time, these schemes are vulnerable to certain denial of service attacks: a 'bad' peer can interfere with the lookup or search processes yet typically cannot easily be identified or avoided. Recent schemes acknowledge this problem and hope to address it by using byzantine fault-tolerant schemes.
- 'Sybil' Attacks: identity replication attacks [19] illustrate that even byzantine fault-tolerant protocols cannot adequately operate in a completely free-for-all environment.

We observe that these problems can be avoided by limiting those who may participate in the peer-to-peer network, either by limiting the system to a closed environment (such as Google's search engine implementation) or by system design (such as Farsite [19] or Oceanstore's inner ring [10]).

This paper discusses the issues in using Mnemosyne to provide a commercial distributed storage service, for both long-term applications (such as archival storage) and short-term storage needs (such as secure messaging applications). For this scenario, we consider Mnemosyne nodes to be "servers" and under the control of either one service provider, or a group of reputable, federated providers (as is the case with BGP peers in the Internet, for example). In contrast, Mnemosyne "clients" are users of the system and do not participate in the peer-to-peer network.

This division is useful for many reasons: servers are readily identifiable (e.g. via a certificate) and so Sybil attacks are avoided; servers are at least somewhat trusted, and so byzantine fault tolerance techniques can be applied, and servers are stable and can be dimensioned so that load skew and server "churn" is not an issue.

Clients are still part of the overall system, and may use the same or similar protocols as servers. In general, clients contact any server and ask it to perform the relevant operation (lookup, routing, etc) as its proxy. The results, if any, are returned by the same server. Clients do not directly communicate with each other.

This generalised many-clients/many-servers model also allows us to draw a clear trust boundary within the system: the point of client-server interaction. As we discuss in section 5, it is where one can use micropayment schemes (e.g. Chaumian digital cash [20]) to not only pay for the service on a per-transaction basis but also mitigate denial of service attacks (by making them expensive).

## 3 A Functional Overview of Mnemosyne

Mnemosyne [1] is a peer-to-peer steganographic storage system built at Sprint Labs. The principle of steganographic storage, proposed in [15], is that users of the system who do not have the required key not only are unable to read the contents of files stored under that key (as with a conventional encrypting file system), but furthermore are unable to determine the existence of files stored under that key.

In a multiuser distributed system such as Mnemosyne, this leads to an interesting property: since users cannot know anything about the location of file blocks stored by other users, it is always possible for them to unwittingly overwrite them; the existence of a file allocation table or list of in-use blocks defeats the steganographic properties of the system. Instead, Mnemosyne uses redundancy in the form of erasure codes to prevent file data being lost due to the write activity of other users.

The process by which a user of Mnemosyne stores a vector of bytes in the system can be broken down into four phases: dispersal, encryption, location, and distribution, which we describe in turn.

**Dispersal:** In the dispersal phase, the data is encoded to make it robust in the face of losses of blocks. Our implementation uses Rabin's Information Dispersal Algorithm [21] in the field $GF(2^{16})$ to transform $n$ blocks of data into $m > n$ blocks, any $n$ of which suffice to recover the original data. In our implementation, $m$ is typically $5n$ for file data (as opposed to directories and inodes), the block size is 1000 bytes, and $n$ is no greater than 32. Files of larger than 32,000 bytes are handled by chunking. We discuss choices of $m$ and $n$ later on in this paper.

**Encryption:** The dispersed blocks from the previous step are now encrypted under the user's key $K$. The purpose of this is twofold: firstly for security and privacy, but secondly for authenticity. This is especially important in a system like Mnemosyne where we expect significant numbers of blocks to be overwritten by other users in normal usage. Thus we need a mechanism by which a user can determine whether a block subsequently retrieved from the network is really the one originally written. Since this check must be made before the data is reconstituted by reversing the dispersal step, encryption is done after dispersal.

Mnemosyne as currently implemented uses the AES algorithm in Offset Code Book (OCB) mode [22] to provide security and a 16-byte Message Authentication Check in one step. An alternative would be two-pass generic composition approaches, but OCB makes for easier key management. AES-OCB encryption of the (padded) dispersed blocks adds 16 bytes of MAC to the message for a total of 1024 bytes per block.

**Location:** Mnemosyne achieves its security properties by storing encrypted data blocks in pseudo-random (and to an adversary, unpredictable) places in a large virtual network store, which is then mapped onto distributed physical storage devices. The locations of the encrypted blocks making up a data set are determined by a sequence of 256-bit values obtained by successively hashing (using SHA256 in the current implementation) an initial value $h_0$.

The initial value $h_0$ depends ultimately on the user's key $K$. For file data itself, $h_0$ is generated randomly and stored in an inode; for inodes and directory blocks, $h_0$ is computed by encrypting the pathname or directory name with the key $K$ and hashing the result.

**Distribution:** The sequence of 256-bit location identifiers from the previous step is finally mapped onto physical storage using a peer-to-peer network of storage nodes, each of which holds a fixed-size physical block store.

For each block to be stored, both the node identifier and the block offset within the block store are derived from the corresponding location identifier. In the current implementation of Mnemosyne, the top 160 bits of this identifier are used to as a node identifier in a Tapestry [6] network. A block to be written is sent to a randomly selected Tapestry node, which routes the block to the "surrogate" node for the 160-bit node identifier. The next 20 bits of the location id are then used as a block number in a 1GB block store.

The node location component of distribution is relatively independent of the underlying peer-to-peer lookup service employed; while Mnemosyne currently uses Tapestry [6], any of [7–9] would work just as well. Indeed, since the Mnemosyne client is not itself a Tapestry node, but communicates with a randomly chosen node using a simple UDP-based protocol, a client could conceivably use several P2P networks from different storage providers simultaneously. What we require of the P2P network is deterministic routing of messages tagged with arbitrary $n$-bit identifiers to nodes.

The block store at each peer-to-peer node supports only the following two operations:

- **putBlock**(*blockid*, *data*)
- **getBlock**(*blockid*) $\rightarrow$ *data*

Note that the block storage nodes themselves need perform no authentication, encryption, access checking, or block allocation to ensure correct functioning of the system, though they might for billing purposes. Indeed, a block store may ignore the above operations entirely: as long as sufficiently many block stores implement the operations faithfully, users' data can be recovered.

**Retrieval:** Data is retrieved from Mnemosyne by the reverse process: given an initial hash value for the data, a user computes the sequence of location identifiers and uses it to retrieve at least $n$ "good" blocks (i.e. blocks which pass the MAC check). Given these, the original data can be recovered by inverting the IDA. Requests for blocks can proceed in parallel to reduce the effects of network latency.

### 3.1 Filing system structures

In [1] we describe one implementation of a per-user filing system over the data storage and retrieval procedures described above. The filing system uses directories and inodes to simplify the management of keys and initial hash values, and also handles versioning of files, a necessity since it data is never actually deleted from Mnemosyne, but rather decays over time. When retrieving blocks for a file it is essential that the blocks retrieved all correspond to the latest version of the file.

### 3.2 Implementation

A working implementation of Mnemosyne for Linux exists. The client is written in C and C++, using freely available reference implementations of SHA-256 and AES-OCB. It provides a command-line interface with operations for key management, creating and listing directories, and copying files between Mnemosyne and the Unix file system. A simple block protocol over UDP is used for communication with block servers. The block server is implemented in Java and runs on Tapestry [6] nodes. Performance is plausible - we can copy files into Mnemosyne at 80 kilobytes per second, and read them at 160 kilobytes per second. A principal limiting factor in both cases is our (unoptimised) $GF(2^{16})$ arithmetic implementation.

We hope to make the source code for our implementation available in the near future.

## 4 Experimental results: measuring file resiliency

In this section we use simulation to investigate the feasibility of Mnemosyne as a serious storage service. While analytical results are obtainable for most of what follows, the simulation results are more useful for giving a feel of how the system works. Since Mnemosyne relies on extensive redundancy in data encoding, and overwrites of blocks are part of the normal operation of the system, we define here two quantities which are of obvious interest when considering whether the system is practical: *efficiency* and *life expectancy*.

### 4.1 Asymptotic Efficiency and Capacity

We define *capacity* of a Mnemosyne system as the quantity of (undispersed) data that can be stored. The *efficiency* is this capacity, expressed as a fraction of the total raw disk space available in the system. Since we are dealing with a multiuser system in which users are generally unaware of each others' activities, efficiency is primarily of interest to service providers as part of their provisioning process: it gives them a handle on how much raw disk space they need to provide.

We first present a naive but intuitive notion of efficiency that we call *asymptotic* efficiency $E_{asym}$. Figure 1 shows the results of repeatedly writing fixed-size files into a store under different coding schemes. In this, as in all the other results in this section, files are 5 blocks in size before dispersal and the total size of the store is 4,000,000 blocks. The simulation keeps track of which files are still recoverable from the store, that is, those files that still have 5 blocks of their original data in the store.

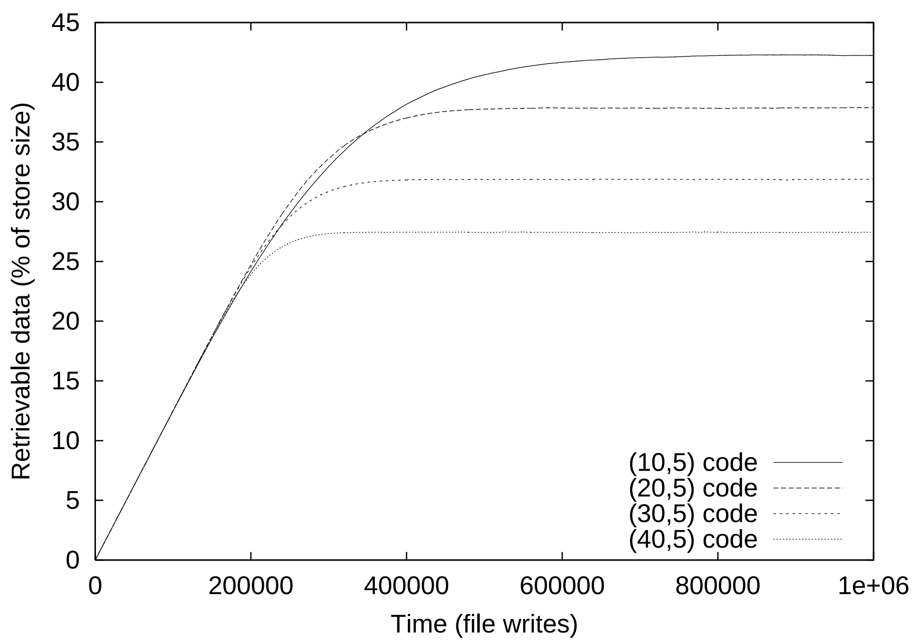


**Fig. 1.** Writing files into a simulated 4Mblock store.

Figure 1 shows the total number of accessible files, starting with an empty store. As more files are written, the system reaches a steady state in which writing each new file, on average, renders one existing file unreadable (by overwriting one of the five remaining blocks of the victim file). The total amount of data in accessible files at this limit is the asymptotic capacity $C_{asym}$ of the store. The asymptotic efficiency $E_{asym}$ is this divided by the capacity of the store (4M blocks).

We can see that $E_{asym}$ varies with the redundancy used: for a redundancy factor of 2—a (10,5) code—we can store data equal in size to about 42% of the store. As we increase the redundancy of the coding, this figure drops to about 27% for a redundancy factor of 8 with a (40,5) code.

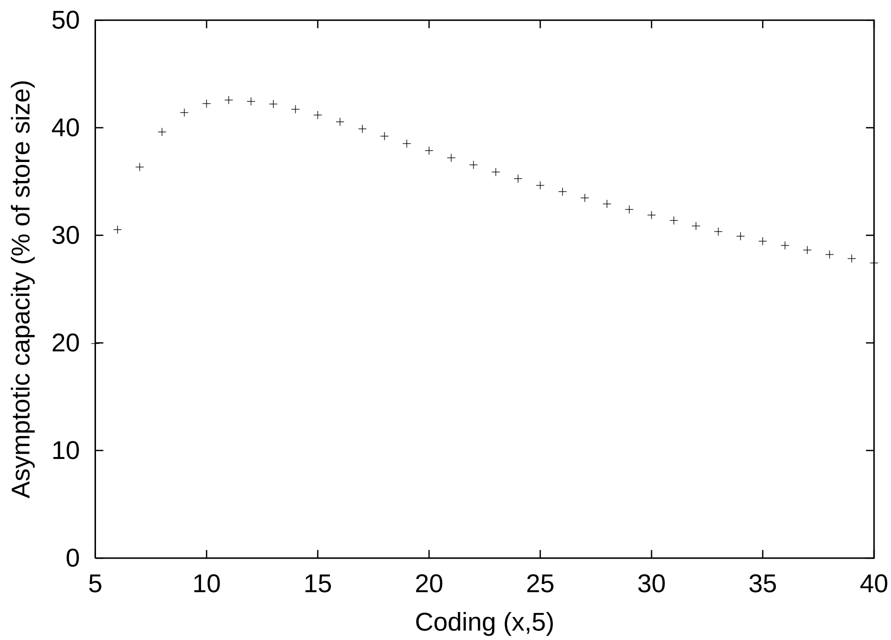


**Fig. 2.** Asymptotic efficiency $E_{asym}$ of a simulated 4Mblock store.

Figure 2 shows how $E_{asym}$ varies with a larger number of coding schemes. While there is a maximum at around (12,5), a more important result is that for redundancy factors between 3 and 8, the asymptotic efficiency of the store remains at over 25%, a remarkably high figure. This suggests that we have, from an efficiency standpoint, a fair degree of freedom in choosing our coding scheme.

While $E_{asym}$ is relatively intuitive measure of store utilisation, it doesn't capture anything about how long files can be expected to survive after being initially written. More precisely, it contains no notion of the distribution of file lifetimes.

### 4.2 Life Expectancy

Figure 3 shows the cumulative distribution of file lifetimes. We run the experiment as before, but now each time a file becomes inaccessible, we record the number of files written between the time when the victim file was first written, and the time it is lost. This distribution of file lifetimes is important to Mnemosyne users because it gives them realistic expectations of how long their data is likely to be around if it isn't "refreshed" (rewritten to the store).

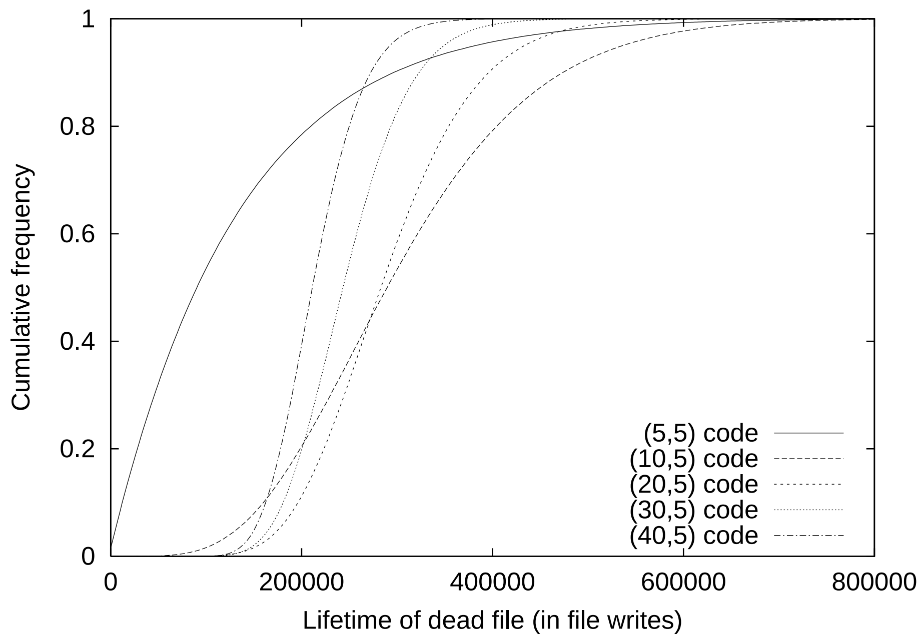


**Fig. 3.** File lifetimes in a simulated 4Mblock store.

For comparison, we have included the (5,5) code in this graph, that is, the results of writing files with no redundancy at all. This clearly doesn't constitute a useful file service: a significant number of files written under this scheme are completely lost within a small number of subsequent file writes.

For redundant coding schemes, the median file lifetime decreases as the redundancy of the coding increases - the halfway point of the curve moves left towards the origin. However, the variation in file lifetimes also decreases.

Of primary concern to a client of Mnemosyne is the region of these curves very close to the $x$-axis, which is the region where the proportion of files lost is very low. Figure 4 shows the region between 0 and 0.01% of files. We can see that for (25,5) and (35,5) coding schemes, the chance that a file will have been lost from this store after 80,000 other files have been written is less than 0.00001. In other words, the chances that a file is accessible after 80,000 other files have been written is better than 99.999%.

Recall that in this simulation we used a store with 4 million blocks, and files were all 5 blocks in size. 80,000 files therefore corresponds to 400,000 blocks, or 10% of the store size in *undispersed* data.

It's clear that this figure of 10% in blocks (rather than files) is valid for other file sizes and store sizes, as long as files are much smaller than the total store size (a reasonable assumption in a distributed storage system of this kind). The implication is that, using a redundancy factor of 5, users can retrieve their files with 99.999% certainty provided they do it before 10% of the total store capacity in raw data has been written by other users.

This probability is likely to be better than the probability of disks or machines failing during a reasonable period, and is unlikely to be a dominant factor in determining the resilience of a storage service. We discuss below some options for allowing users to determine how long it takes for 10% of the store size in real data to be written or, in this case, 50% of the store size in dispersed data to be overwritten. This latter measure is more significant since this is independent of any client's coding scheme, can be directly observed by a service provider, and also estimated by a 3rd party using sampling techniques, as we discuss below.

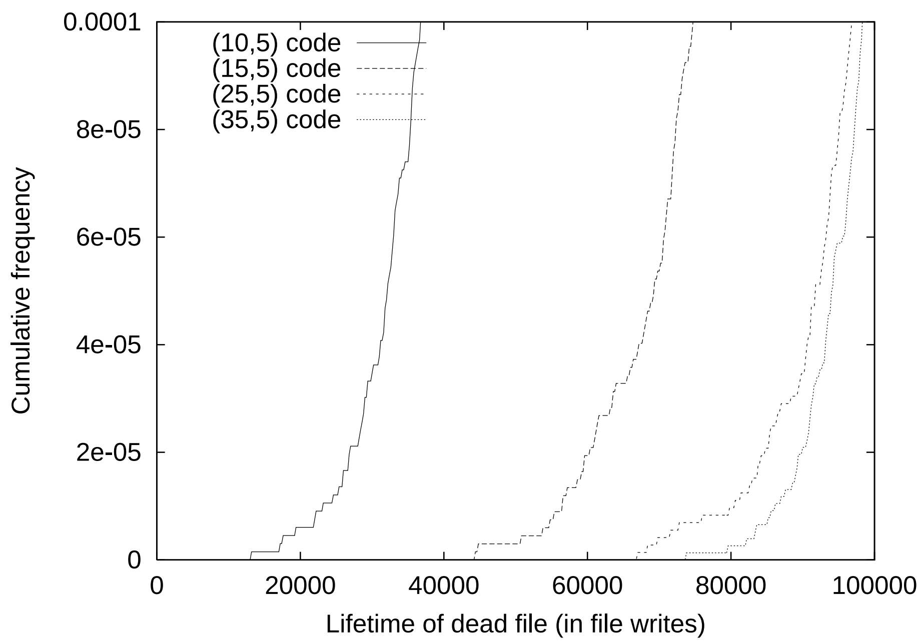


**Fig. 4.** Enlarged area of figure 3.

### 4.3 Effective Capacity and Efficiency

The preceding observations lead to more useful corresponding measures of capacity and efficiency. For a particular probability of retrieval (say 99.999%), a file must be retrieved before some amount of the store $k$ blocks of the store have been has been written to (in our example, 50% of it or 2M blocks). For long term storage, a given file will need to be rewritten ("refreshed") each time this point is reached.

If we consider a store of fixed size where no files are ever discarded, it is clear that no more than $k$ blocks may be used to store data—when $k$ have been used, all further writes must be to refresh existing files in the store.

This gives us the notions of *effective* capacity and efficiency: for a given redundancy of encoding and retrieval probability, the effective capacity $C_{eff}$ is the maximum quantity of raw data that can be written to the store before all subsequent writes must be refreshes. $C_{eff}$ is equal to $k$ divided by the redundancy of the coding scheme used (5 in our example).

The effective efficiency $E_{eff}$ of the store is the ratio of $C_{eff}$ to the total store size. On the basis of our simulations, we can say that the effective efficiency of a Mnemosyne store with 5:1 redundancy and retrieval probability of 99.999% is about 10%. Our current work includes reproducing this result analytically.

10% effective efficiency (i.e. we need 10 times as much disk as the amount of data to be stored) is a plausible degree of overprovisioning. Whether this makes economic sense in practice depends on the ever-changing cost tradeoffs between disk drives, network bandwidth, maintenance, etc. However, in the next section we formulate arguments as to why this extra cost in disk space might be more than compensated for by cost savings in areas like administrative complexity and billing infrastructure.

## 5 Economics of Steganographic Storage

In this section, we discuss charging for steganographic storage (understood now as storage where clients write blocks arbitrarily in a large distributed store). Mnemosyne as described has a number of important characteristics in this regard.

In Mnemosyne, all filing system structures and policy have been moved into the client. While our inode-based filing system works well in practice, each user is free to implement whatever structure they like over the basic block store. Furthermore, tradeoffs involved in how to encode a file, how often to refresh it, etc. are decided by the client and not the server. The server in fact doesn't really know how much capacity a client is using.

The upside of this is that all the complexity associated with traditional storage services (whether network-attached storage systems providing CIFS and NFS, or storage-area networks providing raw SCSI logical units, or object-based storage like Oceanstore) is removed from the server and transferred to the client. Steganographic storage providers *only* need to be concerned about overall capacity planning.

However, charging for such a service cannot therefore be done on the basis of space used, or time periods over which such space is used, since the servers do not have access to this information. It is ultimately unreasonable in Mnemosyne to charge based on the presence of a user's data block on the system, since no guarantees are made as to the life expectancy of a single block.

Instead, the natural charging model for Mnemosyne and other steganographic storage services is to charge *individually per write transaction* (and, possibly, read transactions as well). This several additional advantages: there is no longer any need for a centrally coordinated accounting system to be involved in individual writes: since each transaction can be treated independently, billing records can be aggregated and processed later, off-line.

Indeed, note that in Mnemosyne the service provider has no direct need to know the identity of any client executing a write, as long as their money is good. Consequently, a digital cash scheme such as [20] not only has potentially desirable anonymity properties, but can also result in even lower billing overhead.

### 5.1 The Mnemosyne tradeoff

Of course, it is hard to obtain accurate information for how much of the cost of running a storage service is due to maintaining file system metadata, user account details, enforcing space quotas, and billing. This will also vary between different providers and the different types of storage service offered, and will also change over time as technology and demand evolve.

However, there does seem to be agreement that this complexity is a significant cost factor, and this overhead increases if we consider large numbers of users with (relatively) small storage requirements, rather than the large, corporate-wide data warehousing applications that current storage providers target.

*This is the crucial tradeoff* that Mnemosyne offers: additional requirements in raw disk space, in return for extreme operational simplicity. Whether a Mnemosyne service makes commercial sense depends on the nature of this tradeoff in each particular instance. Our contribution in this paper is to present the Mnemosyne model as a radical alternative to traditional approaches to storage service.

We observe that RAID as a technology for increasing reliability and performance offer very little benefit in our case, since Mnemosyne is already spreading load and redundant data over a much larger set of disks. Using raw disks without RAID controllers will, on a large scale, offer better global optimisation of both performance and reliability.

As an additional comment, we would point out that in scenarios where much disk space would normally be unused (for example, due to static allocation between users), the tradeoff tips more in Mnemosyne's favour.

### 5.2 The Time Constant of a Store

For clients to be able to make effective use of a Mnemosyne store, they need to know the rate (over time) at which writes to the store happen, expressed as a fraction of the total size of the store. This allows them to calculate, for their own desired encoding scheme and retrieval probability, the refresh interval they need to work to. Providers also need this value in order to perform provisioning, discussed below.

This value is the *time constant*, $\tau$ of the store. It can be understood intuitively as the time period over which $1/e$ of the store is written to. In reality $\tau$ is not constant over time, but changes both as load on the store changes and capacity is added.

Given this notion of a time constant, we can now discuss how storage providers and clients make decisions over the service.

### 5.3 The Provider's Standpoint

How does a provider of steganographic storage provision and charge for their storage capacity? The problem is somewhat flexible due to the extra variable of write rate: if the capacity is small, clients will need to refresh their files more often, which increases the write rate, which increases the cost to clients of storing data for long periods. This of course assumes in turn that clients know the rate of block writes expressed as a fraction of the store capacity.

It will become clear that there are parallels between this problem, and that of charging and traffic engineering in data networks. We point out some of these connections below. Several approaches suggest themselves. We outline three (non-exclusive) options here; they are the subject of ongoing research:

**Fixed advertised time constant:** The provider advertises a particular time constant $\tau'$. This $\tau'$ then becomes the basis of the contract (explicit or implied) between clients and provider. The provider needs to ensure that enough disk space is added to the system to ensure that the "real" time constant $\tau$ is always greater than $\tau'$. While very simple, this approach has some parallels with how IP backbone capacity is provisioned today.

**Measurement-based provisioning:** 3rd-party measurement services can verify or discover the time constant of a particular storage provider. They would do this by writing known data into a randomly chosen collection of blocks, and then observing how rapidly this data becomes overwritten. This market-based approach allows clients to select storage providers with appropriate capacity and also encourages storage providers to provision adequately to remain competitive. Naturally such a scheme requires that unscrupulous providers cannot or do not give preferential treatment to measurement services in an attempt to appear to have better time constants than they really do. A similar issue exists today with web latency measurement services like Keynote and content distribution networks.

**Congestion pricing for storage:** Rather than a fixed charge per transaction, a provider might charge for a write transaction based on the current instantaneous write rate seen by that provider. This has obvious parallels with the congestion pricing approach to network provisioning [23]. Most of the proposals related to congestion pricing (such as 3rd party aggregators who charge a premium in return for carrying the risk of price fluctuations) carry over to storage in this case.

### 5.4 The Client's Standpoint

A client using a Mnemosyne service is interested in ensuring a particular level of resilience for their data while minimising cost. The time constant $\tau$ of the store gives, for any encoding scheme, the longest period after which data must be refreshed to be retrievable with the appropriate probability. Assuming a fixed transaction cost, the client should then pick an encoding scheme which minimises the number of writes over a long period within these constraints.

If there are multiple storage services to choose from, the situation becomes a little more complex. Each will have different time constants and charges, and so the client should perform the optimisation described above for each service and pick the cheapest.

## 6 Conclusions

We have presented Mnemosyne, a distributed storage system based on a peer-to-peer lookup service. The system holds no metadata whatsoever at storage nodes (including information about users). Mnemosyne therefore presents a novel tradeoff: simplicity of operations, maintenance and management in exchange for disk space. Mnemosyne has been implemented and demonstrated using the Tapestry routing layer as its lookup service.

This paper has examined the practical considerations for both service providers and users: service providers must provision their service and charge for it, clients must implement their desired resiliency tradeoffs, minimise their costs, and calculate when to periodically refresh files. We have shown that for both players, optimal behaviour depends on the time constant of the store: a measure of how quickly new data is written to it.

## References


1. Steven Hand and Timothy Roscoe. Mnemosyne: Peer-to-Peer Steganographic Storage. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Boston, MA*, March 2002.
2. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.
3. Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, July 2000.
4. Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceeding of the 9th USENIX Security Symposium*, pages 59–72, August 2000.
5. F.S. Annexstein, K.A. Berman, and M. Jovanovic. Latency Effects on Reachability in Large-scale Peer-to-Peer Networks. In *Proceedings Thirteenth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.
6. Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2000.
7. S Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM 2001, San Diego, California, USA.*, August 2001.

8. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001, San Diego, California, USA.*, August 2001.
9. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM Internation Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany*, November 2001.
10. John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.
11. S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.
12. F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Banff, Canada.*, October 2001.
13. Anthony Rowstron and Peter Druschel. Storage management and caching in PAST, a large scale persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Banff, Canada.*, October 2001.
14. Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International Workshop on Networked Group Communications (NGC2001), London, UK*, November 2001.
15. Ross Anderson, Roger Needham, and Adi Shamir. The Steganographic File System. In *IWIH: International Workshop on Information Hiding*, 1998.
16. Matei Ripeanu and Ian Foster. Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Boston, MA*, March 2002.
17. Bryce Wilcox-O'Hearn. Experiences Deploying a Large-Scale Emergent Network. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Boston, MA*, March 2002.
18. Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can Heterogeneity Make Gnutella Scalable? In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Boston, MA*, March 2002.
19. John R. Douceur. The Sybil Attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Boston, MA*, March 2002.
20. D. Chaum, A. Fiat, and M. Naor. Untraceable Electronic Cash (Extended Abstract). In *Advances in Cryptology - CRYPTO '88 Proceedings*, pages 319–327, 1989.
21. M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Communications of the ACM*, 36(2):335–348, April 1989.
22. Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *Eighth ACM Conference on Computer and Communications Security (CCS-8)*. ACM Press, August 2001.

23. Peter Key. Service differentiation: Congestion pricing, brokers and bandwidth futures. In *Proceedings of the Ninth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 1999)*, June 1999.