# Palimpsest: Soft-Capacity Storage for Planetary-Scale Services

Timothy Roscoe
*Intel Research at Berkeley*
*2150 Shattuck Avenue, Suite 1300*
*Berkeley, CA, 94704, USA*
troscoe@intel-research.net

Steven Hand
*University of Cambridge Computer Laboratory*
*15 J.J. Thompson Avenue*
*Cambridge CB3 0FD, UK*
steven.hand@cl.cam.ac.uk

## Abstract

Distributed writable storage systems typically provide NFS-like semantics and unbounded persistence for files. We claim that for planetary-scale distributed services such facilities are unnecessary and impose an unwanted overhead in complexity and ease of management. Furthermore, wide-area services have requirements not met by existing solutions, in particular capacity management and a realistic model for billing and charging.

We argue for ephemeral storage systems which meet these requirements, and present Palimpsest, an early example being deployed on PlanetLab. Palimpsest is small and simple, yet provides soft capacity, congestion-based pricing, automatic reclamation of space, and a security model suitable for a shared storage facility for wide-area services. It does not "fill up" in the way that an ordinary filing system does. In fact, Palimpsest does not arbitrate resources—storage blocks—between clients at all.

## 1 The Need for Ephemeral Storage

In this paper we address the distributed storage needs of wide-area services designed to run over the kind of shared, planetary-scale computational infrastructure envisioned by the PlanetLab[12], Xenoservers[14], WebOS[18] and Denali[20] projects. So-called "persistent data" in such services consists of the (relatively static) code and data which comprises the service, configuration information, output in the form of log files and transaction records, and system state such as checkpoints and scoreboards. This is a different application area than that addressed by filing systems intended for human users.

We observe that *almost all* this data is either ephemeral, or else a current copy of data which is or will shortly be held or archived elsewhere. Nevertheless, it must be stored with high availability for some period of time. For example, checkpoint state need only be kept until the next checkpoint has been performed, but before then is of high value. Similarly, transaction records must be periodically copied offline for processing, but until this happens it should be hard for a system failure or malicious agent to delete the data or obtain unauthorized access to it.

Current practice is to store such data on a disk directly attached to its node, or use a distributed storage service. The former fails to provide availability in the event of a node failure, while the conventional examples of the latter implement unnecessarily strict semantics and fail to address the resource issues involved in sharing the storage resources between many independent wide-area services.

Similarly, file system security requirements for such services differ from traditional time-sharing systems. Often it should be hard to delete the data or for unauthorized entities to access it, but there are no requirements for complex access control arrangements. Likewise, since the "users" of the storage service will be software services themselves, the human factors issues associated with key storage are largely obviated.

On the other hand, a shared, widely distributed environment imposes other requirements and in particular the need to deal gracefully with a shortage of capacity. While it is true that storage in conventional environments (PCs, workstations, LANs) is generally underutilized, we point out that in these cases the storage is administered by a single organization who has an interest in limiting usage or buying new disks when needed. This is not at all the case in a shared, distributed environment where multiple services must be given incentives to release storage resources and share storage space among each other, particularly when that storage is provided by a third party. Furthermore, wide-area storage services "in the wild" must be robust in the face of denial-of-service attacks, including those that use up storage space.

Distributed filing systems which aim at NFS-like semantics or something similar (e.g. [11, 15]) do not meet these requirements, and employ considerable complexity and incur considerable overhead in providing behavior which, while essential in a LAN environment with human users, is inappropriate in our scenario. Furthermore, they do not address the capacity and denial-of-service requirements mentioned above. Some recent wide-area file systems[5, 17] have moved away from an NFS-like model to, for instance, immutable and read-only file service, neither of which is a requirement in our case.

In the rest of this paper, we give an overview of Palimpsest[1], followed by more detail on the central algorithms and architecture. Along the way we try to point out engineering trade-offs and other areas for further investigation. We then discuss charging for storage based on congestion pricing techniques, and end with comparison of Palimpsest to other wide-area file systems.

## 2 An Overview of Palimpsest

Palimpsest is not a mainstream, traditional distributed filing system. It does not implement NFS semantics, does not provide unbounded persistence of data, and implements a security model different from that traditionally derived from multi-user timesharing environments. It does not "fill up" in the way that an ordinary filing system does. In fact, Palimpsest does not arbitrate resources—storage blocks—between clients at all.

Instead, Palimpsest delivers storage with true, decentralized soft-capacity (clients get a share of the overall storage resource based on the current total number of users, and how much each is prepared to pay), and removes the burden of garbage collection and resource arbitration from storage service providers. Data stored by clients in Palimpsest is secure, and persists with high availability for a limited period of time which is dynamically changing but nevertheless predictable by clients. In addition, Palimpsest provides a novel, financially-viable charging model analogous to congestion pricing in networks and highly appropriate for anonymous microbilling, further reducing administrative overhead.

In Palimpsest, writers store data by writing erasure-coded fragments of data chunks into an approximation of a distributed FIFO queue, implemented by a network of block stores. Over time, as more writes come in, fragments move to the end of the queue and are discarded. A reader requests fragments from the nearest set of storage nodes according to some desirable metric, such as latency. The combination of a distributed store and erasure coding provides high availability.

Storage nodes don't care who writes blocks, but instead charge for each operation. Billing can thus be performed using digital cash, and Palimpsest doesn't need to retain any metadata related to user identity at all.

Each storage node maintains a value called its *time constant*, which is a measure of how fast new data is being written to the store and old data discarded. By sampling the time constant from a series of storage nodes (an operation which can be piggybacked on reads and writes), a writer can obtain good estimates of how long its data will persist, even though this value changes over time.

Consequently, Palimpsest does not "fill up"—in normal use, it's always full. Instead, as the load (in writes) increases, the time over which data persists decreases. The problem of space allocation or reclamation thus never arises. Instead, Palimpsest introduces a new trade-off between storage capacity and data persistence. This trade-off allows the application of techniques originally developed for congestion pricing of network bandwidth to be applied to storage, a concept we explore below.

## 3 The Design of Palimpsest

The basic functionality provided by Palimpsest clients is the storage or retrieval of a *chunk* of data. This is a variable length sequence of bytes up to some maximum value (128kB in our current prototype). To store a chunk, it is first erasure-coded (using Rabin's Information Dispersal Algorithm[13]) into a set of $m$ fixed-size fragments (currently 8kB), any $n$ of which suffice to reconstruct the chunk. $n$ is simply the chunk size divided by the fragment size; $m$ can be chosen by the client to give the desired dispersion factor — the trade-off here is between resilience to the loss of fragments versus storage cost and network bandwidth.

The fragments are then authentically encrypted (using the AES in Offset Codebook Mode[16]) to produce a set of fixed size blocks which are then stored at a pseudo-random sequence of locations within a 256-bit virtual address space; a distributed hash table maps the 256-bit virtual block numbers to block identifiers on one of a peer-to-peer network of block stores. To recover the chunk, the pseudo-random sequence is reconstructed and requests made for sufficiently many blocks to reconstitute the chunk.

The pseudo-random sequence is generated by successive encryption under our secret key of a per-chunk initial value, and so provide statistical load-balancing over the storage nodes in the system. It also makes it hard for an adversary to selectively overwrite a file since the block identifiers are unpredictable without the key. Data is already protected against tampering or unauthorized reading by the encryption and MAC. A blanket attempt to overwrite all data is rendered prohibitively expensive by the fact that Palimpsest charges when a block write transaction takes place (see below).

The use of the IDA also means that Palimpsest will tolerate the loss of a fraction of block servers, and also the loss of individual write operations or the failure of a fraction of read operations.

Block servers hold a FIFO queue of fragments which is also indexed by the 256-bit block identifier. When a write request for a block id $b$ is received, $b$ is looked up in the index. If an existing block with id $b$ is found, it is discarded and the new block added to the tail. Otherwise, the head of the queue is discarded and the new block added to the tail. Block stores thus perform no arbitration of resources — there is no “free list”, neither is the block store interested in the identity of the writer of any block.

Block servers also keep track of the arrival rate of block write requests, expressed as a fraction of the size of the server's store[2]. This number is the *time constant* of the server, and is sampled by clients as described below.

This basic chunk storage is the only facility provided by Palimpsest. Filing systems can be implemented above this using standard techniques, but the details of this are entirely up to the application.

## 3.1 Persistence and the Time Constant

As new blocks are written to block stores, “older” blocks are eventually discarded. This will ultimately make any given chunk irretrievable; if this is not desired, clients need to take steps to ensure their data remains live.

This persistence can be attained by *refreshing* chunks from time to time. To aid clients in deciding when to perform refresh operations, each block store keeps track of its *time constant*, $\tau$: the reciprocal of the rate at which writes are arriving at the store as a fraction of the total size of the store.

If our distributed hash table adequately approximates a uniform distribution, each block store will tend to have the same time constant. If the global load on Palimpsest increases, or several block stores are removed from the system, the time constants of remaining stores will decrease. Conversely, if new capacity (in the form of block stores) is added to the system, or the total write load to Palimpsest decreases, the time constants will increase.

Each read or write operation for which a response is obtained includes the value of $\tau$ at the block store when the response was sent. Clients use these values to maintain an exponentially weighted estimate of system $\tau$, $\tau_s$. This quantity can then be used to determine the likely expiration of particular blocks, which may then be used to drive the refresh strategy.

The most straightforward refresh strategy would simply write a chunk, derive an estimated $\tau_s$ based on the responses, wait almost that long, and then rewrite the chunk. To handle unexpected changes in the value $\tau$ after the chunk has been written, clients can periodically sample the block stores. Since blocks are moving through the FIFO queues in the block, $\tau_s$ and consequently the refresh rate can be adjusted in a timely fashion before data is lost.

More sophisticated refresh strategies are an area for research, although we expect at least some will be application-specific.

We note that the value of $\tau$ acts as a single, simple, system-wide metric by which administrators may decide to add more storage capacity. This contrasts with the complex decision typically faced by operators of distributed storage systems.

## 3.2 Concurrent Reads and Writes

Reconstituting a chunk of data stored in Palimpsest introduces some subtle challenges in the presence of concurrent reads and writes to the same chunk. Palimpsest addresses this by guaranteeing that reads of a chunk will always return a consistent chunk if one exists at the time.

When a Palimpsest client requests a block during chunk retrieval, three outcomes are possible:

1. the block server returns the “correct” block (viz. the one which was stored as part of the chunk);
2. the server is unavailable, or does not hold a block with the requested identifier (resulting in an error response or a timeout);
3. a block is returned, but it is not part of the chunk in question.

The third case is not entirely detected by the use of the MAC, since if a writer updates the chunk in place, using the same pseudo-random sequence to write the new blocks, it becomes impossible to tell which *version* of a chunk each block is from. This in turn leads to a situation where a chunk cannot be reconstituted in reasonable time because, although enough valid blocks are theoretically available, a small number are from an older version and “poison” the process of reversing the erasure code.

We could address this problem by attempting to recombine all combinations of fragments which pass the MAC check until we get a reconstituted chunk which passes some other, end-to-end integrity check. Apart from being inelegant, this becomes computationally impractical even with modest chunk sizes and redundancy in coding.

Another approach is to use a different sequence of block ids for each new version of the chunk. This approach was used in the Mnemosyne system due to its desirable steganographic properties, and while effective for small numbers of highly secure files, it has a major disadvantage for a typical Palimpsest scenario: it requires a piece of metadata (the initial hash value of the "current" version) to be held externally to the chunk, but which changes whenever the chunk is updated. This causes a "cascading" effect when one tries to build a hierarchical filing system above Palimpsest's chunk storage: updating a file requires the file's metadata to change, which causes an update to an inode or directory chunk, which causes *its* metadata to change, and so on up to the root.

Consequently, Palimpsest adopts a third approach: including a version identifier in each fragment. This allows a valid chunk to be reconstituted if enough fragments can be retrieved, but a new version of the chunk to be written to the same block sequence without modifying the chunk metadata. Even if the chunk length changes and an inode also needs to be updated, this technique prevents updates cascading up the directory tree.

Version identifiers for chunks need not be sequential, or indeed monotonic. Palimpsest uses the upper 64 bits of the AES-OCB nonce (which is stored with the block) to denote the version of the chunk, and uses voting to decide which fragments should be used to reconstitute the chunk. To retrieve a chunk, some number of fragments is requested and histogrammed according to version identifier. If any histogram bin contains enough fragments to reconstitute the chunk, the algorithm succeeds. If no more fragments can be requested, it fails. Otherwise more fragments are requested.

The derivation of good schedules for retrieval of fragments (how many to request at each stage, how to use network knowledge to choose which fragments to request, etc.) is a field rich in possible trade-offs, and is an open area of research.

How to behave when the above algorithm fails is application specific. For example, if the reader believes the data to be periodically updated, it can immediately retry since the failure may be due to a concurrent write.

## 3.3 Charging and Billing

As we argue in Section 1, storage space is a *scarce* resource in planetary-scale systems: the popular wisdom that storage is "free" only applies in local or small-scale systems. Palimpsest must therefore induce clients to moderate their use of storage capacity without any knowledge of either the global set of clients, or the amount of storage each is using.

We achieve this by using a micropayment scheme in which every `read` and `write` operation must be accompanied by a digital token. A two-tier charging model is used since while `read` operations have only a direct cost to the provider (that of locating and serving the relevant block), `write` operations have both direct and indirect costs: in the steady-state, each `write` will displace exactly one existing data block and with some probability $p$ will render a certain file inaccessible. Hence `write` operations warrant a higher charge than `reads`.

Since a charge is imposed with each use of the system, and since increased use by a user $i$ adversely affects all other users, we can make use of control algorithms based on *congestion pricing* [9]. In general this means that we ascribe to each user $i$ a utility function $u_i(x_i, Y)$ where $x_i$ is the number of `write` operations performed by user $i$ and $Y$ is the "congestion" of the entire system. We need $u_i$ to be a differentiable concave function of $x_i$ and a decreasing concave function of $Y$; this corresponds intuitively to notion that storage in Palimpsest is "elastic": users always gain benefit from writing more frequently, and from less competition from other users.

Of course the incentive to increase one's write-rate is balanced by the cost of paying for it. In traditional congestion pricing schemes (designed for regulating use of network resources), a spot-price is advertised at any point in time; this price is computed so that an increase in demand is balanced by the marginal cost of increasing the resource. In Palimpsest, rather than explicitly increasing the price, we simply advertise the new time constant – for a given level of risk, this will result in a rational client modifying their refresh rate so as maintain the expected data lifetime they require. We believe this improves on the scheme proposed in e.g. [8] since we explicitly model the variation in $\tau$ and predict its future values rather than relying on instantaneous feedback.

As an alternative, a storage service can advertise a guaranteed time constant to its clients. It can then use the time constant measurements made by the block stores to perform the analogue of network traffic engineering: working out when to provision extra capacity, and/or change its advertised service. Note that, under this model, denial of service attacks in Palimpsest are actually an expansion opportunity for the storage provider: maintaining an appropriate $\tau$ can be achieved by increasing the storage resources commensurate with the additional revenue obtained.

As in the case with network congestion pricing, clients which require strong availability guarantees (or, equivalently, predictable prices) can deal with intermediaries who charge a premium in exchange for taking on the risk that a sudden burst of activity will force more refreshes than expected. The provision of strong guarantees above Palimpset then becomes a futures market.

## 3.4 Status

Palimpsest is under active development, and we hope to deploy it as a service on PlanetLab in the near future. The Palimpsest client is implemented in C, while the block server is written in Java and currently uses Tapestry as its DHT.

## 4 Related Work

Palimpsest shares some philosophy with the Internet Backplane Protocol[1], in that the conventional assumption of unbounded duration of storage is discarded. Palimpsest supports similar classes of application to IBP while pushing the argument much further: all storage is "soft capacity", with no *a priori* guarantees on the duration of persistence. In this way, storage services avoid performing any metadata management or resource arbitration, resulting in substantially simpler implementations. Resource management is pushed out to end systems by means of transaction-based charging and periodic sampling of the store's time constant.

The notion of having no explicit delete operation and relying on storage being "reclaimed" was perhaps first observed in the Cambridge File System[2]. Palimpsest shares the goal of avoiding reliance on users to remove files, rather putting the onus on them to refresh what they wish to retain. We avoid the need for an explicit asynchronous garbage collector by simply allowing incoming `write` operations to displace previous blocks.

A number of wide-area storage systems have recently been developed in the peer-to-peer community. Early systems supported read-only operation (e.g. CFS[5], Past[17]) though with high availability. As with distributed publishing systems (e.g. Freenet[4], Free Haven[6] or Publius[19]) we believe these systems complement rather than compete with Palimpsest. Ironically, Palimpsest's ephemeral storage service shares several implementation techniques (such as the use of erasure codes) with very long-term archival schemes like Intermemory[3].

More recently various efforts at Internet-scale read-write storage systems have emerged including Pasta[10], Oceanstore[15], and Ivy[11]. All are far more involved than Palimpsest, with schemes for space allocation or reclamation being either complex (in the first two cases) or non-existent. Palimpsest by contrast incorporates support for storage management at its core but without requiring intricate algorithms or centralized policies.

Some of the ideas in Palimpsest were inspired by the Mnemosyne steganographic filing system[7], though there are notable differences, since Palimpsest is targeted at widely distributed Internet services rather than individual security-conscious users. Palimpsest implements an approximate distributed FIFO queue rather than the random virtual block store in Mnemosyne, which increases both the effective capacity of the block store and predictability of chunk lifetimes. Palimpsest's use of charging, choice of encoding scheme, and selection of block identifiers also reflects its goal of providing a useful facility to planetary-scale services.

## 5 Conclusion

We have argued that emerging wide-area, planetary-scale services have different requirements of distributed storage than the existing models based on human users of time-sharing systems, or network distribution of read-only content. Highly general security policies and unbounded duration of file persistence come with considerable complexity and are largely not required in this area. Furthermore, current systems lack effective mechanisms for allocating scarce storage space among multiple competing users, and a viable economic model for resource provisioning.

We claim there is a role here for storage services which offer bounded duration of files, but provide high availability and security during that time, combined with soft capacity and a congestion-based charging model. Such a system is Palimpsest, which we have described and are in the process of deploying on the PlanetLab infrastructure. The design of Palimpsest allows a number of interesting design choices and lends itself to a lightweight, flexible and secure implementation.

## Notes

[1]A palimpsest is a manuscript on which an earlier text has been effaced and the vellum or parchment reused for another.

[2]Currently, all block servers have the same size store, a decision well-suited to the current PlanetLab hardware configurations. We propose the use of virtual servers as in CFS [5] to allow greater storage capacity on some physical nodes.

## References

[1] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *Proc. of ACM SIGCOMM 2002, Pittsburgh, PA.*, August 2002.

[2] A. Birrell and R. Needham. A universal file server. *IEEE Transactions on Software Engineering*, 5(6):450–453, September 1980.

[3] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries, Berkeley, California*, Aug. 1999.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. 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.

[5] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles, Banff, Canada.*, October 2001.

[6] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, July 2000.

[7] S. Hand and T. Roscoe. Mnemosyne: Peer-to-Peer Steganographic Storage. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, March 2002.

[8] P. Key and D. McAuley. Differential QoS and Pricing in Networks: where flow-control meets game theory. *IEE Proc. Software*, 146(2), March 1999.

[9] J. K. Mackie-Mason and H. R. Varian. Pricing Congestible Network Resources. *IEEE Journal of Selected Areas in Communications*, 13(7):1141–1149, September 1995.

[10] T. D. Moreton, I. A. Pratt, and T. L. Harris. Storage, Mutability and Naming in *Pasta*. In *Proc. of the International Workshop on Peer-to-Peer Computing at Networking 2002, Pisa, Italy.*, May 2002.

[11] A. Muthitachareon, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of the 5th Usenix Symposium on Operating Systems Design and Implementation, Boston, MA.*, December 2002.

[12] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. of the 1st Workshop on Hot Topics in Networks (HotNets-I), Princeton, New Jersey, USA*, October 2002.

[13] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Communications of the ACM*, 36(2):335–348, April 1989.

[14] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accountable execution of untrusted programs. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems (HOTOS-VII)*, pages 136–141, 1999.

[15] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The Oceanstore Prototype. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies, San Francisco, CA.*, March 2003.

[16] P. Rogaway, M. Bellare, J. Black, and T. 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.

[17] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large scale persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating Systems Principles, Banff, Canada.*, October 2001.

[18] A. Vahadat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services for Wide Area Applications. In *Proc. of the Seventh International Symposium on High Performance Distributed Computing, Chicago, IL*, July 1998.

[19] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceeding of the 9th USENIX Security Symposium*, pages 59–72, August 2000.

[20] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. of the 5th Usenix Symposium on Operating Systems Design and Implementation, Boston, MA.*, December 2002.