Storage, Mutability and Naming in *Pasta*

Tim D. Moreton, Ian A. Pratt and Timothy L. Harris

University of Cambridge Computer Laboratory, Cambridge, UK
{tdm25,iap10,tlh20}@cam.ac.uk

Abstract. We outline the design and operation of *Pasta*, a peer-to-peer storage system that provides traditional file system semantics while offering the wide-spread caching and distribution required for publishing networks. *Pasta* allows users to manipulate shared files and folders with strong consistency semantics and to collaboratively organize them in unmanaged decentralized namespaces. Storage quotas regulate consumption and allow the network to offer permanence of content.

1 Introduction

We are developing *Pasta*, a peer-to-peer system that provides a global mutable data store that acts as a file system, an archive store and a publication tool. In this paper we introduce the system and focus on the techniques that it uses to encourage efficient use of storage space, to support automated replication and caching and to enable flexible and decentralized namespace management.

Pasta is intended for wide-area use over a federated group of well-connected peers in the Internet (perhaps located at ISPs or in medium-size organisations). The peers provide storage space to *Pasta* and, in exchange, receive system-wide quota-credits that they can allocate to users. Although diverse in ownership and physical location, the peers share a trusted third-party that holds a limited storage management rôle in the network. We envisage clients integrating *Pasta* with their local file system to a similar extent as other network file systems – while not designed for workloads with a high turnover of temporary data, we expect files held in *Pasta* to be directly accessible by users.

Pasta builds on the Pastry peer-to-peer routing substrate which it uses to pass messages between peers and to select nodes at which to locate or insert data [8]. Pastry provides fault-tolerance and scalability. In particular, a message can be passed between any two nodes in an n node overlay network in $log(n)$ hops, while maintaining $log(n)$ routing table entries on each node. Further, Pastry preferentially routes messages between nearby nodes in order to minimise the distance travelled in the underlying network.

In Section 2, we briefly assess two other peer-to-peer file system projects. Section 3 outlines the structure of our system. We describe data storage, file mutability and a decentralized naming scheme, and introduce an approach to quota management. Finally, Section 4 discusses the direction of our future work on *Pasta* and its implementation context.

2 Related Work

We outline the two concurrent projects that are most similar to *Pasta*; a brief survey of other large-scale distributed file systems may be found in [3].

As with *Pasta*, PAST [9] is storage system that is built over Pastry. However, unlike *Pasta*, files inserted into PAST are immutable. The system can offer strong data persistence to users by enforcing storage quotas through a scheme of smart-cards allocated out-of-band. Storage and retrieval operations are performed at the granularity of whole files rather than through random-access interfaces. No human-readable naming scheme exists by which to reference a stored file: rather, a `fileID` associated with the insertion must be passed by other means.

CFS [3] is implemented over Chord [11], a distributed hash table scheme similar to Pastry. Files are split into fixed-size blocks, which are then distributed to nodes in the network. Storage can be guaranteed for a set time period enabled by per-node storage limits based on IP addresses. Users can store files and arrange them hierarchically in a 'file system', which forms the basis of a per-publisher decentralised namespace. CFS offers coarse-grain file mutability but no means for collaborative update by multiple users.

3 Design

A *Pasta* file system is formed from a network of *storage nodes* that hold data blocks for clients. Each storage node is assigned an asymmetric key pair and, when joining the system, computes its `nodeID` from the SHA-1 hash of the public key. These IDs are used by Pastry to route messages between nodes.

Files are split into variable-sized immutable *data blocks*. Each data block has an associated `blockID`, computed as the SHA-1 hash of the block's contents. A block is stored in the network at the node whose `nodeID` is numerically closest to the `blockID`. We insert, retrieve or withdraw a block by using Pastry to route an appropriate message with the destination key set to its `blockID`. Mutable *index blocks* contain file metadata and are the basis of the naming scheme. Each index organises files hierarchically into folders, and describes each file in terms of the ordered list of blocks from which it is composed.

Clients access *Pasta* through their physically-closest storage node. This exposes operations to browse and manipulate the file system hierarchy and to access data via open/close operations and random-access read/write. All these operations are performed at the storage node on which they are invoked, which requests and processes data blocks and index blocks as necessary.

3.1 Data Storage

In this section we describe how *Pasta* clients divide files into blocks in a way that improves storage utilization and cache performance, then how storage nodes replicate and cache these blocks for high availability and low fetch latency.

We adopt the content-based chunking scheme used by the Low Bandwidth File System as a way of splitting files into blocks [5]. This proceeds by calculating a Rabin fingerprint [6] over a sliding window of 48 bytes at each byte offset within the file. Block boundaries are placed whenever the least significant portion of the fingerprint matches a specified break-mark value. Minimum and maximum block sizes avoid pathological cases.

Since blocks are stored under their SHA-1 content hash, those that are common to multiple files will be held only once. Our own evaluation, backed by that in [5], shows that content-based chunking can significantly increase sharing of blocks between 'similar' files: unlike fixed-size blocking schemes it is tolerant to insertions and deletions within files.

The primary copy of each block is held on the storage node closest in ID space to the block's SHA-1 content hash. As with PAST and CFS, fault tolerance can be controlled by specifying a replication factor, k, causing copies to be placed on the $k-1$ nodes immediately adjacent in the ID space to the primary. Nodes use storage management techniques similar to those presented for PAST [9] to maintain these replicas while the block exists in the network, despite nodes leaving, failing, or joining.

It is highly desirable for blocks to be held on nodes physically close to where they are being requested. This minimises fetch distance for a block and balances the query burden between nodes. Pastry's property of local route convergence [1] means that separate requests for a block from nearby nodes are likely to converge at a node physically near to these while also close in the ID space to the block sought. As such, when a requested block has been obtained, the penultimate node in the lookup path caches a copy before forwarding it toward the client. A frequently requested block will develop cached copies 'drawn out' from its storage nodes along the paths by which requests are being routed.

We anticipate that the sharing introduced by content-based chunking will improve the effectiveness of caching: accesses to one file will benefit from the previous caching of blocks shared with other files. In particular, when a popular file is modified, cache entries associated with blocks that are common between the two versions remain relevant.

3.2 Mutability and Naming

File system metadata is held in mutable *index blocks*. Each index block has an asymmetric key pair, generated by the user when the block is first inserted; the SHA-1 hash of an index's public key determines its `blockID`, and any updates to its contents must be signed by the associated private key. As with SFSRO, a voucher is attached to each index block containing the public key and a secure hash of the current contents [4].

Each index block describes a fragment of a user's namespace. Files and mount-points are arranged within an arbitrary hierarchy of folders, up to the system's maximum block size. This scheme allows file metadata to be modified efficiently by updating a single index block, unlike, for example, CFS where a series of directory and inode blocks must be updated [3]. *Pasta* allows entries in an index block's namespace to be drawn from other index blocks, enabling larger file systems to be constructed either by simply *mounting* subtrees, or by *union mounting* the composite contents of all the index blocks specified.

Files are described as a sorted list of (`file_offset`, `blockID`, `indirect`) tuples which specify that the bytes of the file at `file_offset` can be read from `blockID` as necessary. Large files may use indirect blocks, each of which can contain further indirection entries, forming a tree. *Pasta* adopts a close-to-open consistency model on the basis of each index block. After inserting any new data blocks, the client attempts to reinsert the updated index block as a commit operation. *Pasta* detects and rejects conflicting updates by including the content hash of the block's previous contents in the update request. The client then fetches the updated index block and attempts to merge its modifications by replaying a log of the changes made.

Users may use some out-of-band mechanism to advertise the `blockID` of the root index block of file systems they wish to publish, allowing other users to 'link' to them, as happens on the Web today. Writing to regions of a namespace stored in another user's index block will cause new entries to be created in a locally-owned index so that the other user's files and folders are overlaid. Similarly, overlays can be used to 'delete' files or folders from the local view of another user's namespace. If two or more users choose to import each other's views then collaborative work spaces may be created.

We envisage that 'authorities' on particular topics will emerge and over time be linked together to form a structure akin to a Google or Yahoo *directory*, that most users will choose to have as their own root view that they extend and customize as desired.

Pasta adopts the same caching policy for mutable index blocks as for data blocks in order to distribute file metadata toward clients. To ensure consistency over updates, nodes holding copies of an index block must subscribe to a multicast tree rooted at the primary store node of the block. Currently, this tree propagates cache invalidation messages – we intend to experiment with incremental updates and differentiating between passively cached copies and indexes in active use. An application-level multicast system similar to Scribe is used [10]. When a node caches a block, it routes a 'subscribe' message with key equal to the `blockID`, attaching itself in the multicast tree to the node one hop along the original request path. This is the node from which it obtained the block and is therefore already subscribed.

3.3 Quotas and Accounting

Pasta offers persistent storage: any file inserted is maintained until explicitly withdrawn. System-wide per-user quotas are used to regulate consumption. These are enforced by nodes acting as *principal accountants*: when a user inserts a block each node storing a replica informs the user's accountants – a set of nodes generated by iterated hashing of the user's ID. A quorum protocol provides tolerance to Byzantine accountant failures [2].

Storage nodes are responsible for allocating quotas to users: each node provides the system with a fixed unit of storage and may distribute some portion of this as quota credits (the remainder being used for caching). *Storage node accountants* track which users have been credited by which nodes. This structure removes the trusted authority from ordinary operations.

4 Ongoing Work

We are currently incorporating *Pasta* into the Xenoserver [7] project, which is building a *public infrastructure* for wide-area distributed computing. It provides a low-level customizable execution environment over which users can deploy not just their own applications but also their own operating system instances. *Pasta* will fulfil the rôle of a global file system accessible from all Xenoserver nodes. It will be used to hold user-submitted applications for execution on Xenoservers and also to hold the operating system images and standard components necessary for the platform to operate. This context makes *Pasta*'s provision of flexible namespace management with content-based caching particularly attractive.

We intend to evaluate the effectiveness of content-based chunking using file system traces suited to this scenario. We also wish to explore automated methods of structuring files within *Pasta*. For instance, re-organizing index blocks to reflect common access and update patterns, or exploring different chunking policies to reduce false sharing and reflect known file formats.

We are very grateful to Antony Rowstron of Microsoft Research for the public release of the Pastry simulator, and for his support and insightful comments.

References

1. M. Castro, P. Druschel, Y. Hu and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Submitted for publication.
2. M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. OSDI 1999.*
3. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP 2001.*
4. K. Fu, M. F. Kaashoek and D. Mazières. Fast and secure distributed read-only file system. In *Proc. OSDI 2000.*
5. A. Muthitacharoen, B. Chen and D. Mazières. A Low-bandwidth Network File System. In *Proc. ACM SOSP 2001.*
6. M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
7. D. Reed, I. Pratt, P. Menage, S. Early and N. Stratford. Xenoservers: accounted execution of untrusted code. In *Proc. HotOS 1999.*
8. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001.*
9. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP 2001*, Oct. 2001.
10. A. Rowstron, A.-M. Kermarrec, P. Druschel and M. Castro. Scribe: The design of a large-scale event notification infrastructure. In *Proc. NGC 2001.*
11. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01.*