# Pastis: a Highly-Scalable Multi-User Peer-to-Peer File System

Jean-Michel Busca[1], Fabio Picconi[2], and Pierre Sens[2]

[1] INRIA Rocquencourt
Le Chesnay, France
`jean-michel.busca@inria.fr`
[2] LIP6, Université Paris 6 - CNRS
Paris, France
`[fabio.picconi,pierre.sens]@lip6.fr`

**Abstract.** We introduce Pastis, a completely decentralized multi-user read-write peer-to-peer file system. In Pastis every file is described by a modifiable inode-like structure which contains the addresses of the immutable blocks in which the file contents are stored. All data are stored using the Past distributed hash table (DHT), which we have modified in order to reduce the number of network messages it generates, thus optimizing replica retrieval.
Pastis' design is simple compared to other existing systems, as it does not require complex algorithms like Byzantine-fault tolerant (BFT) replication or a central administrative authority. It is also highly scalable in terms of the number of network nodes and users sharing a given file or portion of the file system. Furthermore, Pastis takes advantage of the fault tolerance and good locality properties of its underlying storage layer, the Past DHT.
We have developed a prototype based on the FreePastry open-source implementation of the Past DHT. We have used this prototype to evaluate several characteristics of our file system design. Supporting the close-to-open consistency model, plus a variant of the read-your-writes model, our prototype shows that Pastis is between 1.4 to 1.8 times slower than NFS. In comparison, Ivy and Oceanstore are between two to three times slower than NFS.

## 1 Introduction

Although many peer-to-peer file systems have been proposed by different research groups during the last few years [1–5], only a handful are designed to scale to hundreds of thousands of nodes and to offer read-write access to a large community of users. Moreover, very few prototypes of these large-scale multi-writer systems exist to this date, and the available experimental data are still very limited.

This research was conducted as part of the GDS project (`http://www.irisa.fr/GDS`) of the ACI MD program, supported by the french Ministry of Research.

One of the reasons for this is that, as the system grows to a very large scale, allowing updates to be made anywhere anytime while maintaining consistency, ensuring security, and achieving good performances is not an easy task. Read-only systems, such as CFS [4], are much easier to design since the time interval between meta-data updates is expected to be relatively high. This allows the extensive use of caching, since cached data are either seldom invalidated or kept until they expire. Security in a read-only system is also quite simple to implement. Digitally signing a single root block with the administrator's private key and using one-way hash functions allows clients to verify the integrity and authenticity of all file system data. Finally, consistency is hardly a problem since only a single user, the administrator, can modify the file system.

Multi-writer designs must face a number of issues not found in read-only systems, such as maintaining consistency between replicas, enforcing access control, guaranteeing that update requests are authenticated and correctly processed, and dealing with conflicting updates.

The Ivy system [3], for instance, stores all file system data in a set of logs using the DHash distributed hash table. Each user has its own log to which he appends his own updates. This eliminates the need of a central serialization point, and provides high security against attacks, including attacks from users who turn out to be malicious, but also limits the number of users (more users means more logs to traverse when reading a file).

Oceanstore [2] uses a completely different approach to handling updates by introducing a centralization point called the primary tier. This set of replicas serialize updates using a Byzantine-fault tolerant (BFT) [6] algorithm. However, BFT is expensive, and primary tier nodes must be highly resilient and well-connected. Although Oceanstore has many features, the system is quite complex, and its centralized design may not be suitable for a community of cooperative users.

With the aim of finding a solution to the shortcomings of these systems we have designed Pastis, a highly-scalable, completely decentralized multi-writer peer-to-peer file system. For every file or directory Pastis keeps an inode object in which the file's metadata are stored. As in the Unix File System, inodes also contain a list of pointers to the data blocks in which the file or directory contents are stored. All blocks are stored using the Past distributed hash table, thus benefiting from the locality properties of both Past and Pastry [7].

Our system is completely decentralized. Security is achieved by signing inodes before inserting them into the Past network. Each inode is stored in a special block called User Certificate Block, or UCB. Data blocks are stored in immutable blocks, called Content-Hash Blocks, the integrity of which can easily be verified. All blocks are replicated in order to ensure fault tolerance and to reduce the impact of network latency.

Finally, we have implemented a prototype written in Java. It runs the FreePastry [8] open source implementation of Past and Pastry. We have modified the original FreePastry's implementation, generalizing the Past *lookup* call to efficiently support one of the consistency models provided by Pastis.

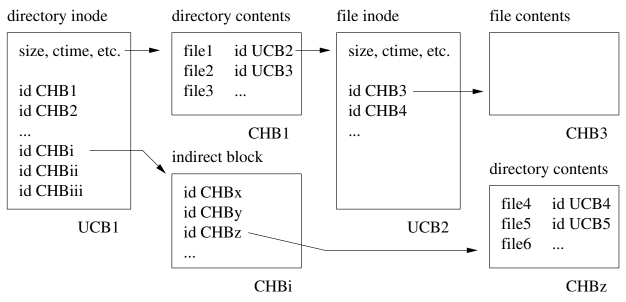


**Fig. 1.** File System Structure

The remaining part of this paper is as follows: section 2 introduces Past and Pastry. Section 3 presents the design of our system in more detail. Section 4 presents our prototype, and evaluate its performance. Finally, section 5 concludes this paper. More details on Pastis can be found in [9].

## 2 Pastry and Past

Pastry [7] is a peer-to-peer key-based routing substrate designed to support a very large number of nodes. In a Pastry network, each node has a unique fixed-length node identifier (nodeid), whose address space can be thought of as a circle ranging from 0 to $2^{160} - 1$. Routing is performed with keys: a message is routed to the node whose nodeid is numerically closest to the specified key.

Pastry's routing algorithm is derived from the work by Plaxton [10]. This type of prefix-based routing can achieve very low hop counts, usually $O(logN)$, where $N$ is the number of overlay nodes. However, one important feature of Pastry is that it takes into account network locality to optimize overlay routes.

Past [11] is a highly-scalable peer-to-peer storage service which provides a distributed hash table abstraction. Past uses Pastry to route messages between Past nodes, thus benefits from Pastry's properties, i.e. scalability, self-organisation, locality, etc. In addition, Past ensures high data availability through the use of replication: it implements a lazy replication protocol that maintains a constant number of replicas of each stored block, as nodes join and leave the network.

Although not detailed in this paper, we have generalized the Past *lookup* call so that it now supports one or more search predicates. This allows us to efficiently retrieve an inode replica whose version stamp is not older than a given value. We use this feature in the read-your-write consistency model, as described in 3.2.

## 3 Design

We begin the description of our file system by presenting how file system data are stored on the network. The data structures used in our design are similar to those of the common Unix file system (UFS). For each file the system stores an inode-like object which contains the file's metadata, much like the information found in a traditional inode.

As shown in Figure 1, each inode is stored in special DHT blocks called User Certificate Blocks (UCBs). For each UCB a private-public key pair is generated by the user who creates the file, i.e. the file's owner, and the private key is stored in the owner's computer.

All inodes contain at least the following information: inode type, file attributes, and security information. This is basically the same information as that returned by the `stat` Unix system call. Specific inode types contain additional fields which are only necessary for the corresponding file type. Regular file and directory inodes, for instance, contain a list of pointers to other blocks in which the file or directory contents are stored. Symbolic link inodes, in turn, contain only the link's destination path.

File and directory contents are stored in fixed-size DHT blocks called Content-Hash Blocks (CHBs). The DHT address of each CHB is obtained from the hash of the block's contents, and is stored within the file's inode block pointer table. As with UFS inodes, we use single, double, and triple-indirect blocks to limit the size of the inode's block pointer table.

In order to optimize directory operations, each directory inode holds a small number of directory entries in the inode itself. Therefore, clients accessing directories that contain only a few files need not retrieve any CHBs. Retrieving or inserting the UCB in which the inode is stored may be sufficient, thereby reducing operation time and increasing performance.

Our design is similar to that of CFS [4], the main difference lying in the use of modifiable blocks (UCBs) to store inodes, thus eliminating the cascade effect of CFS when updating an inode.

### 3.1 Updates and conflicts

Modifying a file or directory in Pastis requires updating the UCB in which its inode is stored, but it also usually involves the insertion of new CHBs. For instance, writing to a file will usually take the following steps:

1. *Fetch the file inode (UCB) from the network*
2. *Fetch the corresponding data block(s) (CHB)*
3. *Modify the data block(s) and insert them into the DHT*
4. *Modify the inode's pointer table with the addresses of the new data block(s), and reinsert the inode into the DHT*

Note that modifying the contents of data blocks changes their DHT keys (which are obtained by hashing the block contents). Thus the need for updating the inode's pointer table.

If two or more clients update an inode concurrently, then a conflict will most probably occur. In our current design, the conflict-resolution mechanism works as follows: each inode update is uniquely identified by a version stamp, consisting of the version number of the update and the unique id of the user who issued it. We define a total order on version stamps by first comparing version numbers, and if they are equal, by comparing user ids.

Each time a user commits an update to a file or directory, the inode's version number is incremented, and the new inode is inserted into the Past DHT. During insertion, the Past client sends the new inode to all replicas. Each replica checks that the new inode's version number is greater than the existing one before the replica is overwritten. If this check fails, the update is aborted.

## 3.2 Consistency

Our system currently supports two consistency models: close-to-open and a variant of the read-your-writes guarantee.

*Close-to-open consistency* [12] is a relaxed consistency model widely employed in distributed file systems such as AFS and NFS. In this model the *open* and *close* operations determine the moment in which files are read from and written to the network. The advantage of using close-to-open consistency is that local write operations need not be propagated to the network until the file is closed. Similarly, once a file has been opened, the local client need not check whether the file has been modified by other remote clients, an operation that would require accessing the network. In other words, the local client can cache the file's contents while it is opened, and keep this cache until the file is closed.

In our system, the close-to-open model is implemented by retrieving the latest inode from network when the file is opened and keeping a cached copy until the file is closed. Any following read requests are satisfied using the cached inode. New CHBs are also buffered locally instead of being inserted immediately into the DHT. Finally, when the file is closed all cached data are flushed to the network and removed from the local buffer.

Note that this scheme works because the immutable data blocks (CHBs) that store the contents of each different version of a given file (a new version appears each time the file is closed) are never removed from the network. If they were, then the data blocks pointed to by a cached inode could be no longer valid. Alternatively, a complex garbage collection mechanism would have to be employed to safely remove unused immutable block from the DHT.

The close-to-open consistency model may be stronger than what many applications actually need. In fact, applications which access files that are seldom shared, or that are not shared at all could benefit from a further relaxed consistency. For these applications we have implemented another consistency model, based on the *read-your-writes* session guarantee, originally introduced by Bayou [13]. Our read-your-writes model guarantees that when an application opens a file, the version of the file that it reads is not older than the version it previously wrote. Once the file is opened, file updates are performed as in the close-to-open model.

The key advantage of the read-your-writes model is that it requires fewer accesses to the DHT than the close-to-open model and thus yields lower response time. Because of possible rollback attacks and Past's lazy replication mechanism, the latter model requires all inode replicas to be retrieved to ensure that the latest version is used. By contrast, in the read-your-writes model, it suffices to retrieve at least one inode replica whose version stamp is not less than a given value. Since all of the replicas are written when closing a file, all replicas normally satisfy the search predicate, including the one that is closest to the application opening the file. We leverage Pastry and Past's locality property and fetch this replica in just one lookup path, using our generalized *lookup* call, thus achieving the lowest possible latency.

Note that in practice, it is highly likely that a file open will retrieve the latest version of the file, as in the close-to-open model. The only case the retrieved inode is not the latest one is when the node queried during lookup is acting maliciously, or has not been updated yet following a recent change in the set of nodes hosting the replicas of the inode.

### 3.3 Security

Pastis ensures write access control and data integrity through the use of standard cryptographic techniques and ACL certificates. Pastis does not currently provide read access control, but users may encrypt files' contents to ensure data confidentiality if needed.

Write access control and data integrity are ensured as follows. The owner of a file issues a write certificate for every user he allows to write to the file. When a user modifies the file, he must properly sign the new version of the inode and provide his write certificate along with the inode. The certificate and the inode signature are checked by DHT nodes before they commit the update. A user performs the same checks when reading the file in order to assert the integrity of the file's contents. These mechanisms along with the use of replication make Pastis tolerant to Byzantine behaviour of DHT nodes and rollback attacks, provided at least one replica is not faulty. However, unlike Ivy, our security model assumes that all users allowed to write to a given file trust one another regarding update operations on that file. Because of space limitations the details of Pastis security mechanisms will not be developed here.

## 4 Prototype and evaluation

The latest version of our prototype uses FreePastry 1.3.2 and runs on any platform that supports the Java VM 5.0.

We developed a discrete event simulator, LS$^3$ [16], in order to conduct experiments on large-scale networks. LS$^3$ simulates such networks by randomly locating nodes on a sphere and deriving network latency from the distance between source and destination nodes. The maximum network latency corresponds to two diametrically opposed points on the sphere.

In order to evaluate the performance of our prototype we use the Andrew Benchmark [12], which consists of five phases: (1) create directories, (2) copy files, (3) read file attributes, (4) read file contents, and (5) run a `make` command. The

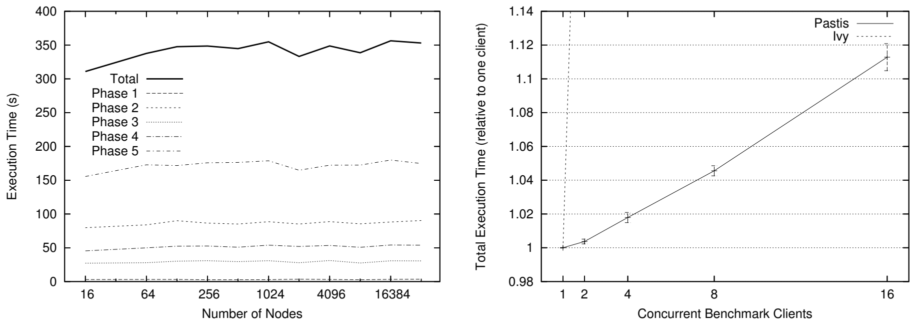


**Fig. 2.** Network size (left), Concurrent clients (right)

source directory we use as input to the benchmark contains two sub-directories and 26 C source and header files, for a total size of 190 Kbytes. The benchmark executes on top of a node running a Pastis/Past/Pastry stack, which in turn communicates with other Past nodes to store and retrieve blocks. These communications can take place either through a real network, or be confined to the local Java VM when using the LS$^3$ simulator.

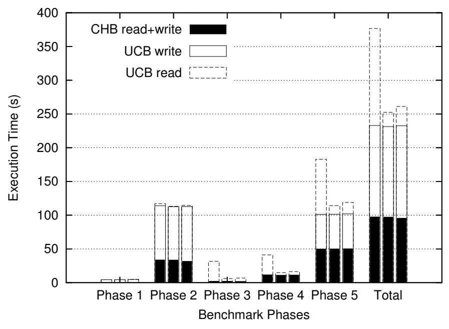
Real experiments are run on Pentiums 4 2.4 GHz with 512 Mbytes of RAM, running Linux 2.4.x. In order to simulate the latency of inter-node communications, we use a DummyNet [14] router, running FreeBSD 4.5. Experiments using the LS$^3$ simulator are run on a Pentium 4 1.8 GHz with 2 Gbytes of RAM.

It is important to notice that all layers from the Pastry layer upwards are unaware of whether they are executing on top a simulated or a real environment. In other words, the executed code corresponding to the DHT and Pastis layers is the same in both cases.

## 4.1 Network size

This experiment evaluates Pastis' scalability with respect to the number of nodes in the network. We run the Andrew Benchmark on a simulated network of increasing size, with a constant maximum network latency of 300ms. We use the close-to-open consistency model, and Past's replication is disabled.

Figure 2 shows the total and per-phase execution time of the benchmark for network sizes ranging from 16 to 32768 nodes. We observe that the total execution time increases only by 13.5% between 16 (311 s) and 32768 nodes (353 s). This good result is mainly due to Pastry's efficient routing algorithm. This experiment confirms, however, that Pastis does not introduce any flaw in the overall design and preserves Pastry and Past's scalability over a wide range of network sizes.



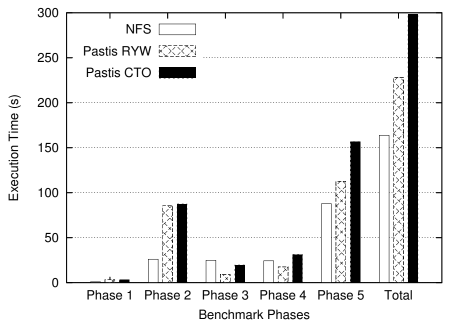


**Fig. 3.** Consistency models (left), NFS comparison (right)

## 4.2 Concurrent clients

In this experiment we evaluate the performance impact of running multiple file system clients concurrently in a real environment. We run from one up to 16 concurrent Andrew Benchmark clients, each client writing to a different directory so that no conflicts are generated. In all cases we use a Past network of 16 nodes, with a block replication factor of 4 and a 100 ms emulated inter-node delay. The consistency model is close-to-open.

Figure 2 shows that the total execution time for a single client is 311 seconds. As the number of clients increases, execution time appears to grow linearly, with only a 2% increase for 4 concurrent clients. This suggests that Pastis scales well in terms of concurrent clients. In comparison, according to [3] an equivalent test performed on the Ivy file system shows a 70% increase when going from 1 to 4 concurrent clients. This is not surprising since having multiple logs (one per participant) forces Ivy clients to traverse all the logs that have been modified, even if the records appended by the other users do not concern the files accessed by the local user. In Pastis, running 16 concurrent client produces only a 11.3% increase compared to a single client, which is very low considering that every node is running a benchmark client.

## 4.3 Consistency models

This experiment compares the performance of the two consistency models that Pastis implements. We run the Andrew Benchmark on a simulated network of 32768 nodes, with a maximum network latency of 300 ms and a block replication factor of 16. We perform three test runs. The first run uses the close-to-open (CTO) consistency model, and the second uses the read-your-writes (RYW) model. The third run also uses the read-your-write model, but this time with 10% of the closest inode replicas being stale.

Figure 3 shows the total and per-phase execution time for each of these three runs. The left, middle and right bars represent the CTO, RYW and RYW with failures runs, respectively. Execution time is broken down into three categories: the lower part of each bar represents the cumulative CHB read and write time, the middle part represents the UCB write time and the upper part represents the UCB read time. We observe that in the close-to-open model, almost 40% of the overall time is spent in UCB reads. This is because all of the live replicas of a given UCB must be retrieved to determine its latest version, as required by the consistency model. As expected, the read-your-writes model yield better performance than the close-to-open model by reducing UCB read time. We observe that while CHB read-write time and UCB write time remain the same as in the close-to-open model, UCB read time decreases by 85% (144 s for close-to-open, 21 s for read-your-writes), yielding a 33% increase in overall performance. Finally, the results also show that even in the presence of 10% stale UCB replicas the overall time increases by only 3% in the read-yrou-write model.

### 4.4 NFS comparison

In this test we compared Pastis' performance to that of NFS v3. This allows us to make an indirect comparison to other peer-to-peer file systems for which a comparison with NFS has been performed [3, 2]. First we run a single Andrew Benchmark client on a real network of 16 machines, each running an instance of Past, with a replication factor of 4. We emulate an inter-node latency of 100 ms using the DummyNet router (a ping between any two machines yields a 200 ms round-trip time). We then run an Andrew Benchmark client on an NFS client accessing a single NFS server, and also emulate a 100 ms latency between client and server (a RPC therefore takes 200 ms).

As shown in Figure 3, total execution time is less than twice that of NFS when Pastis consistency model is set to close-to-open. With the read-your-writes model, Pastis is only 40% slower than NFS. In comparison, other peer-to-peer file systems [3, 2] are between two to three times slower than NFS.

## 5 Conclusion and future work

We have implemented a multi-user read-write peer-to-peer system with good locality and scalability properties. The use of Pastry and a modified version of Past is crucial to achieve a high level of performance, a difficult task since large-scale systems are particularly subject to network latencies.

Another equally important factor is the choice of the consistency model, as strict consistency can impair performance significantly. Therefore, a peer-to-peer file system should offer a range of different degrees of consistency, thus allowing applications to choose between various levels of consistency and performance. Pastis currently provides two relaxed consistency models and future work will involve envisaging and adding new models. Ongoing work focuses on providing support for concurrency control, through the implementation of file locks and exclusive file creation, for application requiring strict consistency.

Finally, our prototype evaluation based on simulation and real execution suggests that Pastis is only 1.4 to 1.8 times slower than NFS. However, our results are still preliminary and must be corroborated by further evaluations.

## References

1. A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
2. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Cambridge, MA, November 2000.
3. A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation* (OSDI 2002).
4. F. Dabek, M. F. 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), Chateau Lake Louise, Banff, Canada, Oct. 2001.
5. Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *5th Symp. on Op. Sys. Design and Implementation (OSDI 2002)*, Boston, MA, USA, December 2002.
6. M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation* (OSDI 1999).
7. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing orlarge-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.
8. FreePastry. `http://freepastry.rice.edu/`
9. J-M. Busca, F. Picconi, P. Sens. Pastis: a highly-scalable multi-user peer-to-peer file system. INRIA Technical Report 5288. August 2004.
10. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA. ACM*, June 1997.
11. A. Rowstron and P. Druschel. Storage management and caching in Past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles (SOSP 2001)*, October 2001.
12. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, volume 6, February 1988.
13. A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B.Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, Dec. 1994.
14. L. Rizzo. Dummynet and Forward Error Correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, June 1998.
15. Pastis. `http://regal.lip6.fr/projects/pastis`
16. LS$^3$. `http://regal.lip6.fr/projects/pastis/ls3`