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

Jean-Michel Busca, Fabio Picconi, Pierre Sens

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Jean-Michel Busca — Fabio Picconi — Pierre Sens

N° 5288

Août 2004

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5288--FR+ENG

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

Jean-Michel Busca* , Fabio Picconi* , Pierre Sens*

Thème COM — Systèmes communicants
Projet Régal

Rapport de recherche n° 5288 — Août 2004 — 29 pages

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 Byz-antine-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.

Key-words: Peer-to-Peer Systems, File Systems, Performances.

* Regal

Pastis : un système de fichier pair-à-pair multi-écrivains hautement scalable

Résumé : Nous présentons Pastis, un système de fichiers pair-à-pair multi-écrivains d'architecture entièrement décentralisée. Dans Pastis, chaque fichier est décrit par une structure modifiable de type inode, contenant l'adresse de blocs non modifiables stockant le contenu du fichier. Les données sont stockées dans la table de hashage distribuée PAST, que nous avons modifiée pour réduire le nombre de messages générés et optimiser la lecture des répliques de blocs. Pastis est simple en comparaison d'autres systèmes existants : il ne requiert ni algorithme complexe tolérant les fautes byzantines, ni autorité administrative centrale. Il est également hautement scalable quant au nombre de nœuds de stockage et au nombre d'utilisateurs se partageant tout ou partie du système de fichiers. De plus, Pastis tire parti des propriétés de localité et de tolérance aux fautes la couche de stockage sous-jacente. Nous avons développé un prototype s'appuyant sur l'implémentation open-source de PAST, et nous avons utilisé ce prototype pour évaluer différentes caractéristiques de notre architecture. Supportant le modèle de cohérence close-to-open, ainsi qu'une variante du modèle read-your-writes, notre prototype montre que Pastis est entre 1,4 et 1,8 fois plus lent que NFS. En comparaison, Ivy et OceanStore sont entre deux et trois fois plus lent que NFS.

Mots-clés : Systèmes pair-à-pair, Systèmes de fichiers, Performances.

1 Introduction

Although many peer-to-peer file systems have been proposed by different research groups during the last few years [4, 9, 10, 13, 17, 19], 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.

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 [17], 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 [13], for instance, stores all file system data in a set of logs using the DHash distributed hash table. In Ivy each update is stored by appending a record to a log. Since records are never removed from the logs, every client has access to all the file system history, which greatly simplifies conflict detection and resolution. Furthermore, each Ivy user has its own log to which she appends her own updates. This has two advantages: first, writes are fast since there is no central serialization point (like Oceanstore's primary tier), and second, data cannot be overwritten by a malicious user since only the log's owner can append data to it. However, as the number of users sharing a given file system increases, the number of logs that need to be traversed to satisfy a read operation also becomes larger, thus increasing network traffic. Although the number of DHash servers can grow to hundreds of thousands, the number of Ivy users sharing a given file does not scale. Another problem in Ivy is that applications have little control over the consistency of data. Although Ivy uses a consistency model similar to close-to-open consistency, applications cannot fully decide when written data are propagated to the network (this is due to the lack of a CLOSE RPC in the NFS v3 protocol specification).

Oceanstore [10] uses a completely different approach to handling updates by introducing some degree of centralization. A primary tier of nodes uses a Byzantine-fault tolerant (BFT) [14] algorithm to serialize all file system updates coming from secondary tier nodes. Since BFT is quite expensive, primary tier nodes must be highly resilient nodes located in high-bandwidth areas of the network. Oceanstore's designers assume that these nodes will be set up and maintained by a commercial service provider. Thus, Oceanstore may not be suitable for a community of cooperative users wishing to use a system which does not depend on a centralized authority. Oceanstore also takes into account network locality to optimize replica

location. An introspection layer provides information about the network conditions, allowing the system to dynamically adapt itself to the current environment. Locality management is absent in Ivy.

Pangaea [19] differs from Ivy and Oceanstore in that it does not rely on a key-based routing layer. Instead, object location is achieved by maintaining a graph of live replicas through which updates are propagated by flooding a special message called harbinger. Moreover, a replica of a file or directory is created on each client that accesses the file. Although this reduces read latency, it can generate an important amount of traffic when updates are propagated.

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 [2].

Our system is completely decentralized. Security is ach-ieved 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 immut-able Content-Hash Blocks so that their integrity can be easily verified. All blocks are replicated in order to improve fault tolerance and to reduce the impact of network latency.

The lack of a central serialization point means that conflicts must be detected and solved in a distributed manner. This can achieved using version vectors and keeping old versions of the inodes so that conflicting operations can be later undone. At the time of the writing of this document we have only implemented a conflict resolution scheme based on the last-writer-wins rule. A more robust conflict resolution mechanism is left for future work.

We have implemented a prototype written in Java. It runs on a modified version of the FreePastry [6] open source implementation of Past and Pastry. We have modified the original FreePastry, generalizing the *lookup* Past call so that one or more predicates can be specified by the application. This allows us to efficiently retrieve an inode replica whose timestamp is not older than a given value.

This paper makes the following contributions. It introduces a completely decentralized multi-writer peer-to-peer file system that supports an arbitrary number of users. It describes several optimizations of the Past DHT that can increase its performance by reducing network communication. It presents the results of our prototype's evaluation using several configurations and test scenarios.

The remaining part of this paper is as follows: section 2 describes some general concepts common to all structured peer-to-peer networks and introduces Past and Pastry. Section 3 presents the design of our system in more detail. Section 4 presents our prototype and some information about our simulator. In section 5 we discuss the results of our prototype evaluation. Finally, section 6 presents related work and section 7 concludes this paper.

2 Structured networks

Most of the recent efforts aimed at designing efficient, highly-scalable file storage systems are oriented towards the use of structured peer-to-peer networks. Although there are some systems based on unstructured or loosely structured networks (e.g., Freenet [4]), they lack some of the key characteristics of highly structured networks, namely deterministic routing and high scalability.

On the other hand, structured networks guarantee message delivery and achieve a high routing efficiency (i.e., a low hop count). This makes them suitable for building a number of services on top of them, such as a distributed hash table or a multicast service. Since these services can be made quite reliable, they can be used in turn to build complex applications such as a distributed file system.

The research community has been working to define the abstractions provided by structured peer-to-peer networks. Three layers, called tiers, have been defined in [3]. Tier 0 offers abstractions related to the routing of messages, also known as key-based routing. Tier 1 services use the abstractions provided by tier 0, and provide their own abstractions to tier 2 layers. Three different tier 1 services have been defined: distributed hash table (DHT), group multicast or unicast (CAST), and decentralized object location and routing (DOLR).

2.1 Key-based routing and DHTs

The lowest level abstraction defined for structured peer-to-peer networks is known as key-based routing (KBR). This type of routing differs from the traditional routing (such as IP routing) in that the destination node is usually not known by the sender. In fact, rather than a destination address KBR messages contain a routing key. According to the definitions in [3], given a key K and a message M, the KBR layer will forward M towards a unique live node known as the root of the key. The root of the key is usually the node whose identifier is numerically closest to the key among all live nodes.

The key's root may change over time as the network conditions change. If the key's current root fails, for example, then another node becomes the key's root. Similarly, a node may cease to be the root of a key K as new nodes join the network. In Pastry, for instance, a joining node becomes the root of all keys for which its node identifier is numerically closest to those keys among all live nodes. Hence the definition of n-root, which is a generalization of a key's root [3]: a node is an r-root for a key if that node becomes the root for the key after all i-roots with $i < r$ have failed.

This definition makes all i-roots with $i \leq k$ the natural nodes for a tier 1 storage service to place the k replicas of a given object. In fact, a message sent by a client to retrieve a replica of a given object will always reach one of the k roots provided all k-roots have not failed simultaneously (and the system has handled node arrivals as well). This ensures that at least one live replica will be found.

At the next layer one of the three defined services is the distributed hash table (DHT) abstraction, which is basically equivalent to a traditional hash table: it maps keys to values. However, in this case by value it is meant an arbitrary object which is often replicated and

stored persistently on one or more different nodes (usually on all the object's i-roots with $i \leq k$). Applications can then retrieve a copy of the object by providing the key under which the object was inserted.

The use of replication is desirable because of its well-known advantages, i.e. increasing fault resilience and masking network latency. However, if the inserted objects are modifiable (which is necessarily the case in a read-write storage system), it introduces the problem of replica consistency. In order to deal with this, the DHT interface may contain a call which allows applications the retrieval of all or a certain number of replicas of an object. As an optimization aimed at preserving bandwidth, the DHT service can first provide the application with some metadata of each existing replica. The application can then decide which replica is to be retrieved from the network, e.g. that corresponding to the latest version of the object.

DHTs also differ from traditional hash tables in that they must take into account a great number of security issues. This is necessary because of the inherently unsafe nature of the resources used by peer-to-peer systems, i.e. the Internet, the computers running the P2P software, etc. The minimum security guarantee that a DHT service must provide to applications is the integrity of the objects stored by it. This is achieved by employing cryptographic techniques such as one-way hash functions and digital signatures.

2.2 Pastry and Past

Pastry [2] is a self-organizing, fault-tolerant decentralized object location and 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) which is randomly assigned when it joins the network. The nodeid space can be thought of as a circle ranging from 0 to $2^{idlen} - 1$, where $idlen$ is the nodeid length in bits.

In order to route a message through the network a Pastry application supplies a key associated to that message. Here the key space is the same as the nodeid space. The routing algorithm then routes the message to the node whose nodeid is numerically closest to the supplied key (i.e. to the root of the key).

Pastry's routing algorithm is derived from the work by Plaxton et al. [1]. The basic idea behind the algorithm is the following: both nodeids and the routing key are interpreted as a sequence of base 2^b digits, where b is a configuration parameter with a typical value 4. Let the length of the shared prefix between a nodeid and a key be the number of the most significant base 2^b digits that are equal in both the nodeid and the key. When a message is routed, each hop forwards the message to a node whose nodeid shares a larger prefix with the routing key. If such a node is not known at the current hop, then a node is selected whose shared prefix has the same length, but which is numerically closer to the key than that of the forwarding node.

In order to achieve good locality properties, Pastry ensures that routing table entries are populated with nodes that are close to the local node according to the chosen metric. This means that when messages are routed, they will follow a path that is optimized ac-

cording to the proximity metric. In addition to this, Pastry locality properties also allow the optimization of replica retrieval in Past [8].

The details of how routing table entries are chosen go beyond the scope of this paper. It will be sufficient to say that whenever a node enters the network it retrieves the neighbourhood sets of a number of other nodes, and uses this information to optimize the locality properties of its own routing table. A detailed description of the algorithm can be found in [2].

Pastry's routing algorithm is highly efficient. If the routing tables are accurate (i.e., in the absence of recent node failures), then the number of routing hops will be with very high probability no greater than $log_{2^b}N$, where N is the number of nodes in the network and b a configuration parameter. With $N = 10^6$ and $b = 4$, this expected hop count is 5, whereas with $N = 10^9$ it takes the value 8. Furthermore, the maximum number of populated routing table entries is equal to $(log_{2^b}N)(2^b - 1)$, which yields 120 entries for $N = 10^9$ and $b = 4$.

These advantages do not come without a cost. Maintaining routing information is expensive, as it requires that a certain amount of messages be exchanged when a node either joins or leaves the network. The number of messages exchanged when a node joins the network is $O(log_{2^b}N)$. Furthermore, the correctness of routing information can only be guaranteed provided the number of concurrent node failures does not exceed $L/2$, where L may have a typical value of 16 or 32 (L is the size of Pastry's Leafset).

Past [8] is a highly-scalable peer-to-peer storage service. It provides applications with a distributed hash table abstraction, which places it at tier 1 according to the terminology presented above. Past uses Pastry to route messages between Past nodes, and in doing so is leveraged by Pastry's properties, i.e. scalability, self-organisation, locality, etc.

Past makes extensive use of replication, which is complemented by caching if the inserted blocks are immutable. The location of the replicas is determined by the key associated to the inserted object. An object replicated k times is usually stored in all the key's i-roots with $i \leq k$. This ensures that at least one replica is always reachable provided all i-roots have not failed simultaneously. This is very unlikely since i-roots should exhibit a very low fault correlation, given that nodeids are randomly assigned throughout the network.

In a peer-to-peer network nodes can be expected to be quite heterogeneous as to the storage capacity they provide to the network. This is a potential source of load imbalance, which is why Past employs a technique known as replica diversion to achieve good load balance properties. This consists in placing a given replica on a node other than one of the key's i-roots, while leaving a pointer to the diverted replica location on the corresponding i-root. A thorough description can be found in [8].

As we pointed out above, when a new node joins the network and becomes the n-root with $n < k$ for a given key, a replica of the object stored under that key must be created on the newly arrived node. Transferring all the necessary replicas at the time of the join would be extremely expensive in both bandwidth and time. In order to avoid this problem, Past creates a pointer to an object replica on the new node so that a replica can be found when it is requested. Then, it transfers lazily all the replicas in the background.

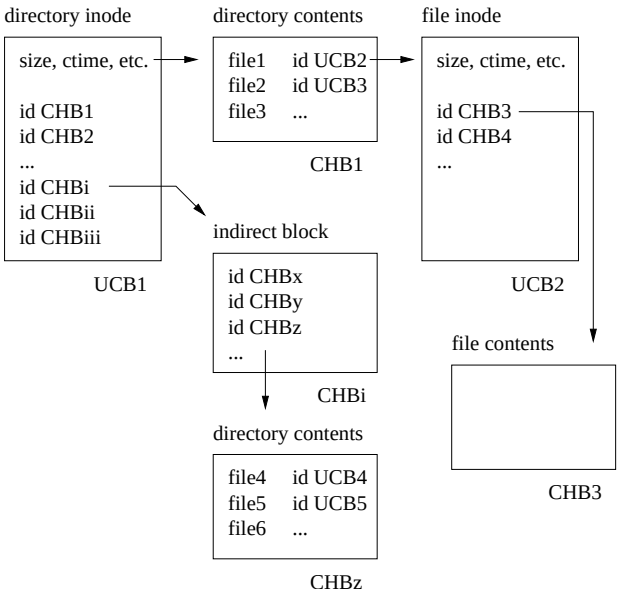

Figure 1: File System Structure

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. In order to avoid confusion, henceforth we will refer by *inode* to our own inode-like object.

As shown in Figure 1, each inode is stored usign a User Certificate Block, or UCB (see section 3.4). For each UCB a private-public key pair is generated by the user who creates the file (i.e., the owner), and the private key is stored in the owner's computer. Although an inode's contents are very similar to that of a UFS inode, not all inodes are equal. For each file type, i.e. regular file, directory, symbolic link, etc., a different type of inode is used.

All inodes contain at least the following information: inode type, file attributes, and security information. File attributes are basically the same as those returned by the `stat` Unix system call, although the *uid* and *gid* fields are missing. Instead, information regarding file ownership and access permissions is encapsulated in what we call the security information. This allows for a more generic identification and access rights semantics than the standard Unix uid/gid/mode scheme (e.g., access control lists). 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 blocks which resemble those of a Unix block device. However, these blocks are immutable and stored in Past Content-Hash Blocks (CHBs). The address of each block 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. Our design is very similar to that of CFS [17], the main difference lying on the fact that CFS inodes are read-only and therefore stored using Content-Hash Blocks.

The contents of a directory are stored in the same way as those of a regular file. Each directory inode points to a set of CHBs containing the directory entries. Each directory CHB contains a fixed number of entries, each entry consisting of a file name and the Past address of the corresponding inode, which may be a regular file, directory, symbolic link, etc.

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.

3.1 Updates and conflicts

As we mentioned in the previous section, our file system structures are similar to those of CFS. However, in CFS each time an inode is modified the file system owner must recalculate the hashes of all directories from that inode up to the root. He must then digitally sign and insert the new root inode. Furthermore, in CFS only the file system owner can update the file system as only she knows the root block's private key. In our design, we choose to use modifiable blocks (UCBs) to store inodes, thus eliminating the cascade effect of CFS's inode modification.

Modifying a file or directory in our system requires updating the UCB in which its inode is stored, but it also usually involves the insertion of new CHBs. Let us consider the case of a write operation to a regular file. Let the write offset be less than the end-of-file, and the length of the data to be written less than the size of a data block. In order to update the file, the system must perform the following actions:

1. *Fetch the file inode from the network*
2. *Fetch the data block(s) corresponding to the offset range [start offset, end offset]*
3. *Modify the data block(s) overwriting the data located at the specified offset range*
4. *Insert the new data block(s) into the network. Since data blocks are stored in CHBs, the hash of the block contents must be recalculated in order to determine the new DHT key(s)*
5. *Update the inode's block pointer table with the addresses of the new block(s). If these addresses are stored using indirect blocks then the indirect blocks must also be modified, which involves recalculating the hash of their contents and inserting the new blocks into the network*

6. Update the UCB with the new version of the inode

Note that each time a file is modified, new immutable data blocks reflecting the newly written data are inserted into the DHT. However, old immutable block are not removed from the network. This implies that the simple fact of overwriting a file makes the file system continuously grow in size. Our system is not the only one to avoid reclaiming storage [13], and as we shall see in the following section we can benefit from keeping immutable blocks when implementing a relaxed consistency model.

If two or more clients update an inode concurrently, then a conflict will most probably occur. Our current design supports only a very simple conflict-resolution scheme based on the last-writer-wins rule for file conflicts. This scheme will be replaced by a more complex one as we will include new features, such as exclusive file creation, in our design.

The current mechanism works as follows: each time a client generates a new version of an inode, it timestamps the inode using the client's local clock. The value of this timestamp is actually that of the ctime field of the inode's attributes. Next, the client inserts the new version of the inode into the Past network by calling the insert Past call. The Past service then sends a message containing the new inode to every live replica, so that their contents can be updated. However, each replica first checks that the timestamp of the new inode is greater than the existing one before it overwrites the existing replica. If this check fails, the inode is considered to originate from a conflicting write, and the update is not performed.

Note that the decision of whether an inode will be overwritten or not depends on timestamps which are generated by different clients. Therefore, for this mechanism to work correctly client should loosely synchronize their clocks using NTP or some other clock synchronization protocol. A similar mechanism is employed by the Pangaea system [19].

This simple conflict resolution scheme has two serious limitations: first, loosely synchronized clocks is not a realistic requirement for a global scale peer-to-peer system, and second, directory conflicts cannot be automatically solved. We are currently reviewing our design to adopt a more robust conflict detection and resolution mechanism, such as one that uses version vectors and allows for automatic resolution by keeping old inode versions.

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 [7, 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 distant 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 data block addresses pointed to by the cached inode. When data are written to a file the locally cached copy of the inode is updated to reflect the new data. Immutable blocks containing the file contents are also buffered rather than inserted into the network in order to avoid delaying the write operation. Finally, when the file is closed all cached data are flushed to the network and removed from the local cache.

Note that this scheme works because the immutable data blocks 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. It focuses on the fact that files can be written by different distant clients, hence the need for synchronization points (open and close operations) in which consistency is guaranteed. 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. Session guarantees were introduced by Bayou [18] as a solution to the problem of providing applications data which is consistent with their own previous actions. An example of this is a program that writes some data to a file, and later reads the file at the same offset only to discover that the previously written data are not there. This behaviour can occur if the read operation retrieves the data from a replica which has not been updated yet.

There are two advantages to using session guarantees: first, they set up consistency-keeping rules on a per-process basis, and second, these guarantees concern only the process, i.e. the instance of a given application, that requires them. In other words, a given file system client may serve many local applications, each one requiring different session guarantees according to their particular needs.

Our enhanced read-your-writes model ensures that read operations always reflects all previous local writes, even if the file is closed and later reopened. However, it provides better consistency than the original Bayou guarantee by making clients retrieve a close replica of the file's inode each time the file is opened, and by propagating writes when it is closed. Therefore, it is highly likely that a file open will reflect any distant writes propagated after a file has been closed. On the worst case, the application will only see its own previous operations, but will not be aware of any distant writes.

The implementation of our enhanced read-your-writes models is as follows: each file system client keeps a table of inode timestamps, which correspond to the latest version of each inode that the client has seen. When a file is reopened, the client retrieves a replica of the inode from the network, making sure that the timestamp of the retrieved inode is not older than that contained in its timestamp table. This procedure consists of two phases: during the first phase, the client instructs Past to retrieve a single replica of the inode.

Given Past's locality properties, it is highly likely that Past will fetch the replica which is closest to the client. This first phase is therefore relatively cheap in network terms. If the timestamp of the retrieved inode is greater or equal than that stored in the client's table, then the replica is valid and the procedure has completed. Otherwise, a second phase is executed in which the client retrieves all inode replicas and keeps that with the most recent timestamp, which should be greater or equal than the one stored in the table. This second phase is the same as that used when using the close-to-open consistency model during a file open.

Since a file close updates all replicas of the file's inode, it is highly probable that the first phase will retrieve a replica that is valid, and that also reflects writes performed by distant clients. This model will therefore produce better performance than close-to-open provided the number of accesses that require the two phases remains small.

3.3 Past modification

As we saw in the previous section, retrieving an inode replica with the enhanced read-your-writes guarantee may involve one or two phases. Since we use the FreePastry 1.3.2 implementation, the first phase makes use the *lookup* call, whereas the second phase employs the *lookupHandles* and *fetch* calls. During the first phase, Past routes a request message so that as soon as a replica is found the message is returned to the client along with the found replica. In the second phase Past retrieves the handles of all live replicas (a Handlesconsists of the replica's address and a portion of the block's contents). The application then decides which replica is to be retrieved (using the partial data contained in the handles), and issues a fetch call to retrieve the full block.

The modification we introduced allows applications to specify a constraint over the block's metadata when performing a *lookup* call. In this way, if the first replica that is found does not meet the required criteria, the message continues its path until another valid replica is found, or it goes back to the client producing a "valid replica not found" error. In this case, the client falls back to the *lookupHandles*+*fetch* method to find a suitable replica.

We use the new *lookup* call to specify a constraint on the inode's timestamp when retrieving an inode replica during the first phase of the enhanced read-your-writes model. This increases the probability that a valid replica will be found without resorting to the more expensive second phase, thus improving performance.

We also introduced two optimizations in Past's implementation of the *lookupHandles* and *insert* calls. The *lookup-Handle* call returns a Handlesfor each of the replicas of the block passed as parameter. It executes in two phases: during the first phase, the root node of the block is queried to determine the list of nodes currently holding replicas of the block, which we call the *replicas set*, and the response is returned back to the caller. In the second phase, the caller queries each of the nodes in the returned set to retrieve the Handlesof the replica it holds. Similarly, the *insert* call involves two phases: the caller first acquires the replicas set of the block to be updated, and then contacts each of the nodes in the returned set to update its copy of the block.

The first optimization we introduced relates to the *lookup-Handle* call only. It consists in having the root node of the block directly forward the request for replica handles to the nodes in the replicas set, thus saving the return trip to the caller for the nodes that are farther than the root node. It is important to note, however, that the standard *lookupHandles* call guarantees by construction that the handles it returns are sorted in ascending order with respect to proximity. This property allows the application to call *fetch* on the closest replica of the block, when it finds that all replicas are equivalent. The modified *lookupHandles* call does not provide this guarantee, and thus must be used in combination with some mechanism for determinig the distance between nodes, as for instance a mechanism based on IP address prefixes.

The second optimization we introduced relates to both the *lookupHandles* and *insert* calls. It consists in caching the replicas set of a given block, and using it as a hint to these calls so that they bypass the first phase of execution. Since file access patterns exhibit temporal locallity, there is a high probability that the replicas set does not change between the time a file is opened for writting and the time it is closed, or even between two successive opens occuring within a short time frame. The modified *lookupHandles* and *insert* calls, however, check that the replicas set they are passed is still up-to-date: when the node which considers itself the current root of the block is contacted to perform the requested operation, it piggy-backs in the reply its current view of the replicas set. Upon reception, the caller checks whether the set has changed, and if so, stores the new set in its local cache and restarts the call using the new set.

Note that both optimizations apply to UCB only; CHB are retrieved from Past by using the *fetch* call, not the *lookupHandles* call. Moreover, CHB are inserted in Past only once since they are immutable, which defeats the caching of their replicas set.

3.4 Security

As we pointed out above, every inode in the file system is stored in a special block called a User Certificate Block, or UCB. A UCB is stored in Past under the public half of the private-public key pair which the owner of the file generated when she created the file. The security scheme which Pastis implements is as follows. Each user of the file system has a private-public key pair, and the owner of a file delivers a write certificate to each user she allows to write to the file. A certificate is thus of the form:

$$\{1{:}\ K_puser,\ 2{:}\ K_pinode,\ 3{:}\ expiration,\ \mathrm{Sign}(1{+}2{+}3,\ K_sinode)\}$$

and it proves that the user whose public key is K_puser has write permission to the inode whose public key is K_pinode, until the specified expiration date. Certificates are signed using the inode's private key K_sinode. Since this key is known only to the user that created the file (i.e., the file's owner), only she can generate valid certificates. Note that the owner of a file can remove write permission from a given user by not issuing a new certificate for that user after the current certificate has expired.

In order to update a file, a user must sign the UCB reflecting the file's changes with her own private key, and provide a certificate which proves that she has write permission on that file. Thus, determining whether a UCB is valid or not requires two signature verifications: first, the certificate provided along with the UCB is authentified using the storage key of the UCB K_pinode, and its expiration date is checked. Then, the UCB's integrity and authenticity is asserted using the user public key K_puser mentioned in the certificate. This verifications are carried out by any Past node which is about to locally store the UCB during an insert operation, as well as by Pastis clients after having retrieved a UCB's replica from the Past DHT. User Certificate Blocks are always sent and stored along with the corresponding certificate so that their verification does not generate further network accesses to find the certificate.

Note that the above discussion only considers write access control. Read access control is not ensured by Pastis itself. As in the Ivy file system, users wanting to prevent the contents of their files from being disclosed should encrypt data themselves before insertion.

Finally, our design also provides protection against Byzantine faults. Byzantine nodes are those which having been subject to a malicious attack or because of software bugs exhibit arbitrary behaviour. If a Past node becomes Byzantine-faulty it may deny the existence of a previously stored block, or return an old version upon reception of a *lookup* message (this is known as rollback attack). However, Byzantine nodes cannot compromise the integrity of a UCB unless they come to possession of the UCB's private key (which would allow them to forge certificates), or of the private key of a user which has write permission. Note that the integrity of CHBs cannot be compromised once their DHT key is known to be valid.

In the presence of a rollback attack, a Pastis client will still behave correctly provided it can retrieve an inode's replica which satisfies the consistency model. When using close-to-open all inode replicas are retrieved on a file open, meaning that the number of Byzantine nodes must be less than k, the replication factor, to guarantee correct behavior. When using the read-your-writes model, the client will either keep the first replica that is found, or retrieve all replicas during the second lookup phase. In either case, a valid inode replica will be found provided less than k replicas are faulty.

4 Prototype

Our prototype is written entirely in Java 1.4, and thus runs on any platform that supports the JVM 1.4. Figure 2 shows a diagram of the different software components. All blocks are coded in Java and are executed within the same Java VM. A core module contains an asynchronous interface, which contains procedures which resemble those of NFS v3, and a block handling module. The latter uses a block cache module, as well as a security manager which performs the cryptography and key handling functions, such as signing a UCB, determining the block key of a CHB, and verifying a block's integrity. Finally, the Past service is accessed through a generic DHT layer which provides the basic DHT abstractions and hides the specified implementation details of the Past interface of FreePastry 1.3.2. This

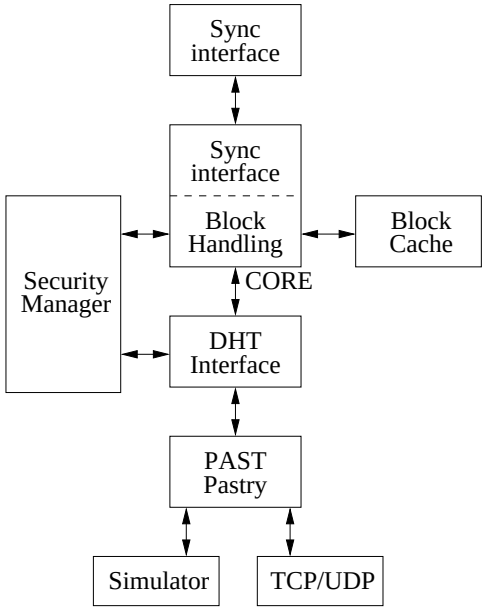

Figure 2: Prototype Architecture

layer also uses the security module in order to verify a block's integrity after it is retrieved from the Past network.

At the bottom of the stack, the Pastry service relies on either the standard Java RMI / TCP / UDP transports, or a simulator used for large-scale experiments. Finally, at the top of the stack we implemented a layer that transforms the asynchronous interface provided by the core module into a synchronous stateful interface for applications. This interface, called *FileAccess* is a hybrid between Java's *File* and *RandomFileAccess* classes. This interface was used by our Java benchmark program for our tests.

There reasons for which we have not implemented an NFS v3 interface is that the *commit* procedure does not correctly convey close information, since it may be called without the file being closed by the application. Since both our consistency models are based on the *open* and *close* calls, a proprietary interface that includes both calls gives the file system client all the information it needs to implement our consistency models.

We developped a discrete event simulator, LS^3, in order to conduct experiments on large-scale configurations. LS^3 represents each Past node of the simulated network with an in-memory object, in which it records the state of the node and its current virtual time. Sending a message to a node amounts to schedule a *reception* event containing the destination node, the message to deliver and the time of delivery. Events are enqueued in a central event

schedule and processed in time-of-occurence order. When a message is delivered to a node, the node's current virtual time is set to the message's time of delivery.

To be as accurate as possible, LS^3 takes three execution parameters into account: network latency, delays induced by network software, and processing time at the application level. Network latency is set to be proportional to the distance between the source and the destination nodes in physical space. We use two kinds of topologies: the *flat* topology, in which all nodes are equally distant from one another, and the *sphere* topology, in which nodes are randomly located on a sphere. Every time a node sends or receives a message, its virtual time is incremented by a small value, which is proportional to the size of the message. This is to reflect the processing of the message by network software on end computers, including message's serialization and deserialization, processing which does not take place in the simulator. When a node completes the processing of a message, the simulator adjusts its virtual time to reflect the real time elapsed since the node received the message. This is to simulate the time it would take in the real world to process the message on a computer with similar characteristics as the one used for the simulation.

It is important to notice that all layers from the generic DHT upwards are unaware of whether they are executing on top a simulated or a real environment. In other words, the executed code corresponding to tier 3 (i.e., client core, cache, security manager, etc.) is the same in both cases.

5 Evaluation

In order to evaluate the performance of our prototype we implemented a Java program that generates a pattern of file system accesses equivalent to that of an Andrew Benchmark [7]. Our benchmark program consists of five phases: (1) create directories, (2) copy files, (3) read file attributes, (4) read file contents, and (5) simulate a `make` command.

The reason for which we implemented the Andrew Benchmark as a Java application is that our current prototype implementation only provides a Java interface to the file system. This is not a problem for the first four phases of the benchmark, but we had to implement a make simulation for the fifth phase. We also included an option in our Java benchmark program that allows us to run the same benchmark but redirecting file access to a local directory instead of our peer-to-peer file system. We can thus compare our system to NFS by specifying a local directory which is actually an NFS mount point. Local accesses are transformed by the kernel into NFS RPCs directed to the NFS server of our choice.

Unless otherwise stated, we ran all our tests using a single instance of the Andrew Benchmark program. The source directory we used as input to benchmark contains two subdirectories and 26 C source and header files, for a total size of 190 Kb. The application accesses the local file system client, which in turn runs on top of a local Past/Pastry node. This node then communicates with other Past instances to store and retrieve Past blocks. As shown in Figure 2, these communications can take place either through an emulated network, or be confined to the local Java VM when using the LS^3 simulator.

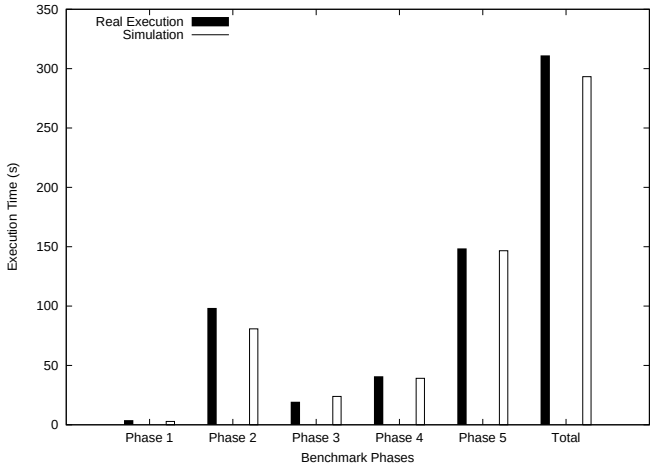

Figure 3: Simulation vs. Real Execution on 16 Nodes

Experiments related to simulation vs. real execution (see subsection 5.1) and concurrent clients (see subsection 5.4) are run on Pentiums 4 2.4 GHz with 512 Mbytes of RAM, running Linux 2.4.x, except for the DummyNet router which runs FreeBSD 4.5. All other experiments are simulations run on a Pentium 4 1.8 GHz with 2 Gbytes of RAM. Note that this latter machine being slower explains the difference we observe on execution times between Figure 3 and Figure 4 for similar network configurations.

5.1 Simulation vs. real execution

The first test that we carried out aimed at determining to which extent the execution of our Andrew Benchmark using the LS^3 simulator differs from an equivalent test using real Pastry nodes. In this configuration we use 16 machines located in the same LAN. Each machine is configured to route IP packets through a DummyNet [5] router, which adds a fixed delay of 100 ms to each packet. Each machine hosts a Past node which communicates with the other nodes using the Pastry WIRE protocol, and Past's replication factor is set to 4. LS^3 is configured to simulate a network with the the same characteristics, and we run the Andrew Benchmark both on the real and simulated networks.

The results are presented in Figure 3. Notice that the total run-times of both benchmarks differ in approximately 10%, the simulated execution being slightly faster than the one using real Pastry nodes. There are two factors that may be causing this difference. First, the model LS^3 uses in order to simulate message processing delay in network layers is too simple to describe Pastry's behaviour, which repeatedly switches between UDP and TCP sockets in

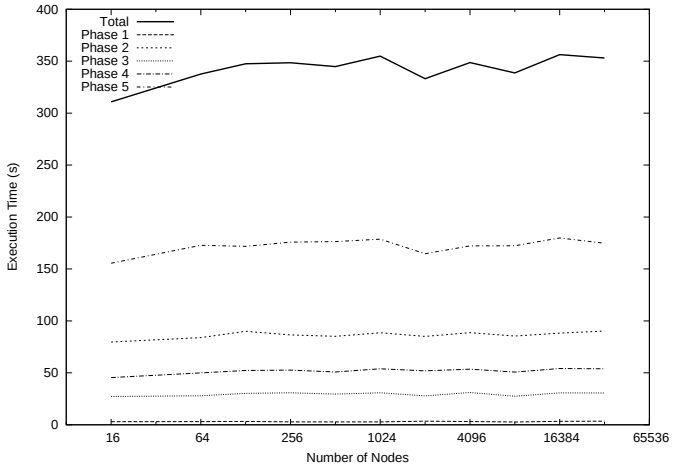

Figure 4: Network Size

order to adapt to network traffic. Second, the resolution of the clock we use to adjust nodes' virtual time (1 ms) may be too coarse to yield accurate results.

In the following sections we make the assumption that this 10% difference between real execution and simulation results will remain constant as we increase the number of nodes in the network. Our hypothesis, however, may not necessarily hold, and we shall therefore consider the benchmark results presented below as preliminary work to be confirmed later by further evaluations.

5.2 Network size

After validating the simulator, we measured Pastis' scalability with respect to the number of nodes in the network. We ran the benchmark on simulated networks of various size, using the sphere topology. The maximal network latency, which corresponds to two diametrically opposed points, was set to of 300ms. We used the close-to-open consistency model, and Past's replication was disabled.

Figure 4 shows the total and per-phase execution time of the benchmark for network sizes ranging from 16 to 32 768 nodes, averaged over four runs. We observe that the total execution time increases only by 13.5% between 16 (311 s) and 32 768 nodes (353 s).

This good result is due to Pastry's routing algorithm, which takes at most $log_{2^b} N$ hops, where N is the number of nodes, to route a message between any two nodes. This experiment confirms, however, that Pastis does not introduce any flaw in the overall design and preserve Pastry and Past's scalability over a wide range of network sizes.

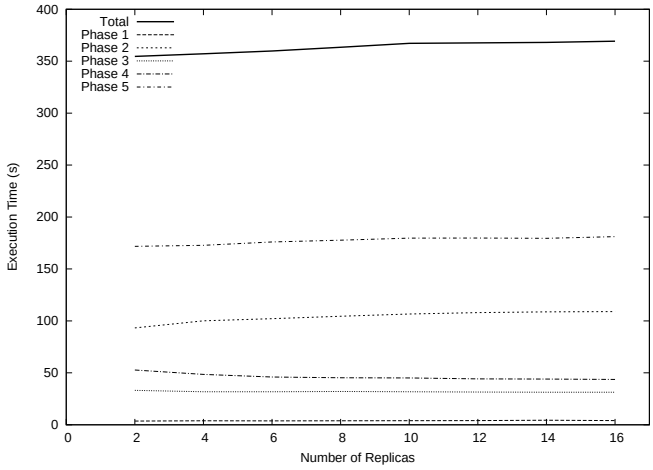

Figure 5: Replication Factor

5.3 Replication factor

Pastis' performance depends on how close block replicas are to the client. By increasing Past's replication factor k, we increase replicas's dispersion. On one hand, it raises the probability that one of the replicas be closer to the client, thus improving read access time. On the other hand, it raises the probability that a replica be farther to the client, which has the counter effect on write operations. Moreover, increasing the replication factor generates more network traffic and puts heavier processing load on nodes.

In order to measure the impact of these opposite effects, we ran the Andrew Benchmark with different replication factors on a network of 32 768 nodes. The other settings were the same as in the previous experiment. Figure 5 shows the total and per-phase execution time of the benchmark for replication factors ranging from 2 to 16.

We observe that the total execution time approximately follows a linear law, increasing by only 4% between replication factors 2 (354 s) and 16 (369 s). Figure 6 takes a closer look at phase 2 (file write) and phase 4 (file read) execution time. As expected, write time increases with the replication factor, although block replicas are inserted in parallel. It increases by 16 s (+17%) between replication factors 2 (93 s) and 16 (109 s), first steeply between replication factors 2 and 6 (50% of the increase), then more gradually. We notice the opposite effect on read time, which decreases by 9 s (-17%) between replica factors 2 (52.5 s) and 16 (43.5 s). Again, the decrease is steep in the first place between replication factors 2 and 6 (50% of the decrease), and then more gradual.

Figure 7 explains these observations. It plots the distribution of access time to block replicas as a function of the replication factor for the network configuration and source directory we used in the experiment, as measured after the benchmark completes. For

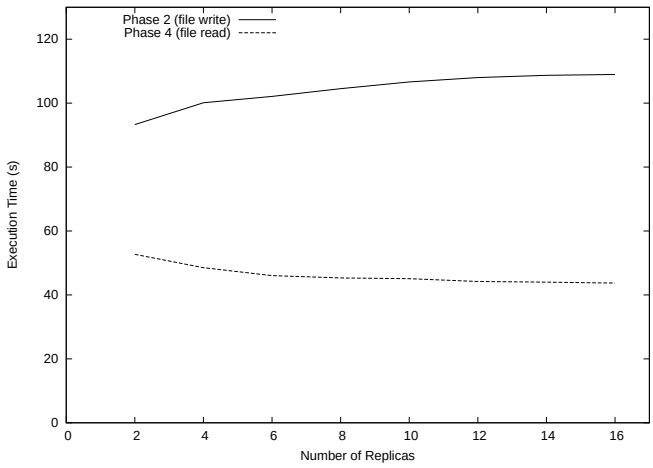

Figure 6: Replication Factor - Detail

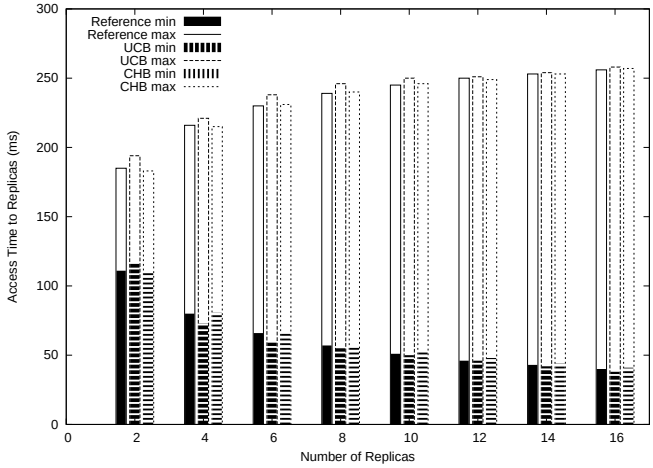

Figure 7: Distribution of Block Replicas

each replication factor, three distributions are depitected: the middle and right bars relate respectively to the UCB and the CHB generated by the benchmark. The left bar correspond to an hypothetical set of blocks, each of which rooted at a specific node of our configuration. This latter distribution serves as a reference point and helps in controling that the UCB and CHB distributions are not biased. The lower part of each bar represents the average

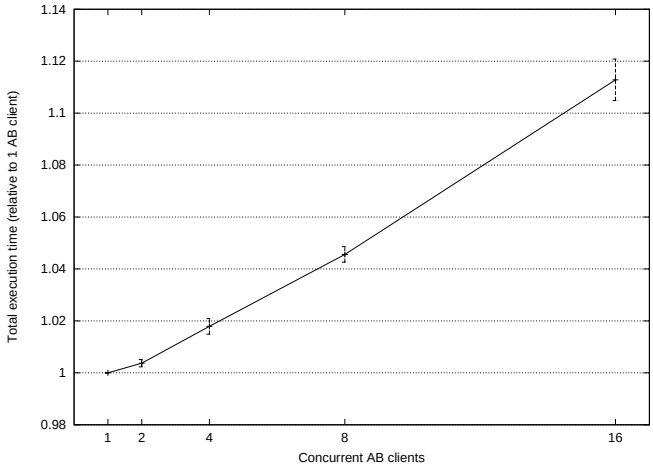

Figure 8: Concurrent clients

access time to the closest replica of the set, while the upper part represents the average access time to the farthest replica of the set. UCB distributions slightly differ from reference distributions because the number of UCB is rather small (29), but CHB distributions closely matche reference distributions. In both cases, we notice that the average closest distance decreases as the replication factor increases, and conversely for the average farthest distance, and that both distributions have similar shape as phase 2 and phase 4 execution time in Figure 6.

5.4 Concurrent clients

In this test we evaluate the performance impact of running multiple file system clients concurrently in a real environment. We ran 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 $k = 4$, and a 100 ms inter-node delay. The consistency model is close-to-open.

Figure 8 shows that the total runtime for a single client is 311.2 seconds. As the number of clients increases, runtime 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 [13] 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

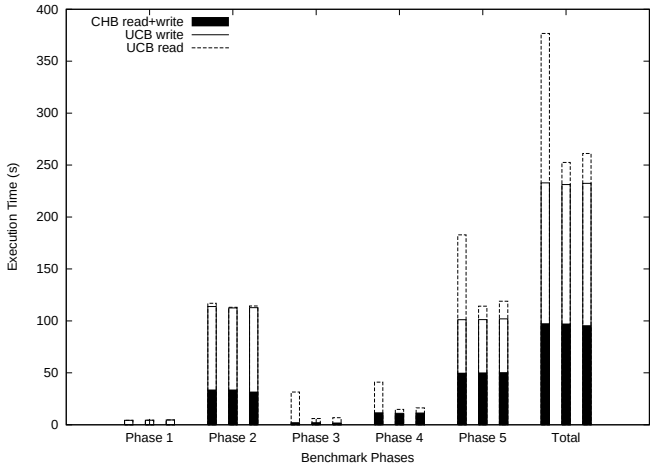

Figure 9: Consistency Models

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.

5.5 Consistency models

Pastis's performance depends on the consistency model that is selected to access files. In order to compare the performance of the two consistency models which Pastis currently implements, we ran the Andrew Benchmark on a simulated network of 32 768 nodes. We used the sphere network topology, with a maximum network latency of 300 ms and a replication factor of 16.

We performed three test runs. In the first we used the close-to-open (CTO) consistency model, and in the second we activated the read-your-writes (RYW) model. In the third run we also used the read-your-write model, but this time we made 10% of the closest inode replicas be stale. This simulates an execution in which the nodes holding these replicas would be down when inodes are updated in phase 2, and would be up again when the file system client must retrieve an inode replica in phases 3-5.

Figure 9 shows the total and per-phase execution time for each of these three runs. The left, middle and right bars represent the close-to-open, read-your-writes and read-your-write with failures runs, respectively. The execution time is split into three categories: the lower part of each bar represents the cumulated CHB read (*fetch* call) and write (*insert* call) time, the middle part represents the UCB write time (*insert* call) and the upper part represents the UCB read time.

We observe that in the close-to-open model, almost 75% of the overall time is spent in UCB reads and writes. There are two reasons for this. First, although a file is represented by one UCB and more than one CHB on average, CHB are written in parallel when closing a file. Second, determining the latest version of a UCB, as implied by the consistency model, requires a *lookupHandles* call and a *fetch* call. As a result, UCB read time alone amounts to almost 40% of the overall time.

As expected, the read-your-writes model yield better performance than the close-to-open model by reducing UCB read time. Finding a UCB replica newer than the last written by the client only requires a *lookup* call in the general case. 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 the presence of stale UCB replicas the overall time increases by 3% in the read-yrou-write model. The reason is that some *lookup* calls fail to retrieve an appropriate replica, which forces the client to resort to the slower *lookupHandles* + *fetch* mechanism.

5.6 Past optimizations

After having compared the performance of the consistency models, we measured the impact of the optimizations we introduced in Past (see section 3.3). We used the same network configuration as in the previous experiment.

We performed five test runs. In the first we used as a reference point the close-to-open consistency model without any optimization. Then we activated in the second run the fast *lookupHandles* call, which directly forwards lookup requests to the replicas set, and the caching of the replicas set for *lookupHandles* and *insert* calls in the third run. The fourth run reproduces the same conditions as the third one, except that we simulated the fact that cached replicas sets were stale in 10% of the calls. Finally, we combined both optimizations in the fifth run, again with a ratio of 10% of stale replicas sets.

Figure 10 shows the total and per-phase execution time for each of these five runs in order. We observe that using fast *lookupHandles* call yields a 23% decrease of UCB read time, which corresponds to a 9% increase of the overall performance. Note that we assumed in this test that we can rely on some mechanism to sort nodes according to network latency. Other experiments, not reported in the figure, show that wihout such mechanism, the gain obtained with the fast *lookupHandles* is almost overriden by longer *fetch* calls.

As expected, the caching of replicas sets improves both UCB read time and UCB write time, by 38% and 30% respectively, which corresponds to a 25% gain on overall performance. It is worth noting that this optimization yield lower performance (282 s execution time) than the read-your-writes consistency model (252 s execution time), but the test is run with the close-to-open model, which is more strict.

The fourth run shows an increase of UCB read and UCB write time in the presence of stale replicas set in the cache. This is because the *lookupHandles* and *insert* calls have to

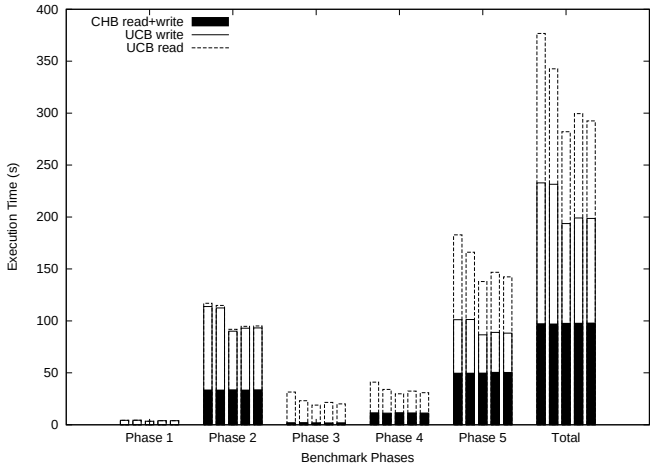

Figure 10: Past Optimizations

be restarted using the up-to-date replicas set which was found. The fifth run shows that the 14% increase in UCB read time can be lowered to 7% by using the fast *lookupHandles* call.

5.7 NFS comparison

In this test we compare 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 [13, 10]. 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 a WAN 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). Results are shown in Figure 11.

When Pastis consistency model is set to close-to-open, total execution time is less than twice that of NFS. With the read-your-writes model, Pastis is only 40% slower than NFS. In comparison, other peer-to-peer file systems [13, 10] are between two to three times slower than NFS.

A more detailed analysis of phase 5 shows that the NFS client took 87.8 seconds to complete the phase, issuing 412 RPCs. The time consumed by network communications is then 412 * 200 ms = 82.4 seconds. More than 80% of these RPCs are GETATTRs and LOOKUPs, the remaining RPCs being 17 CREATEs, 42 WRITEs, and 17 COMMITs (17 files are created during phase 5). No READ RPCs are generated since the file contents are available in the NFS cache. On the other hand, we can measure Pastis' network accesses by

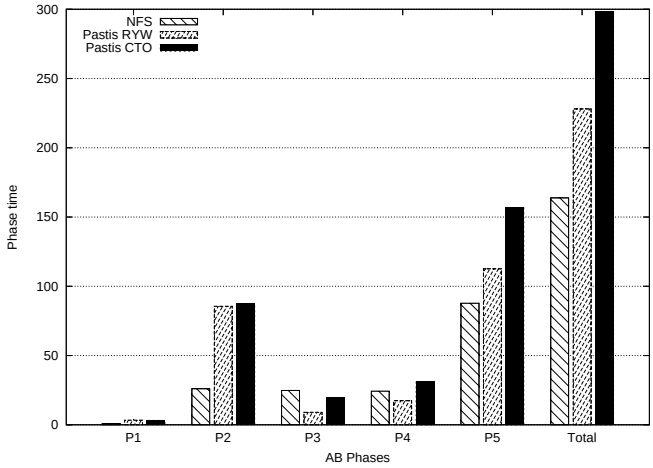

Figure 11: NFS comparison

looking at the number of DHT operations generated by the file system client. During Pastis' execution of phase 5, which took 173 seconds, the client inserted 51 UCBs, with an average Past insert time of 820 ms. The number 51, which is three times 17, is explained by the fact that two UCBs are inserted at a file creation (one UCB for the file and one for the directory in which it is located), and that the file's UCB is reinserted after a file close. Other accesses to the Past DHT include 92 CHB inserts (average time 949 ms), 58 lookups (312 ms), 106 fetchs (376 ms) and 106 lookuphandles (411 ms). It is worth noting than some inserts are issued in parallel (e.g., CHBs containing file data that are inserted when the file is closed).

6 Related work

In this section we list some peer-to-peer file systems designs that have been proposed by different research groups.

Ivy [13] is a distributed multi-user read-write file system. To the user, Ivy behaves much like an NFS file system, providing consistency semantics similar to the close-to-open model provided the network is not partitioned. Update serialization in Ivy is easily accomplished by traversing all logs of a file system. Since clients insert updates into their own logs, no central Byzantine-fault tolerant point is need to commit updates. However, Ivy's one-log-per-participant design poses a limit on the number of users that a file system may have. Ivy relies on the DHash [17] and Chord to store all file system data and route request between nodes.

Oceanstore [10] is a global-scale peer-to-peer persistent storage system. Its designers envisage a global commercial storage service maintained by a number of provider companies,

which would offer storage space to users in exchange for a monthly fee. The storage space is provided by the companies themselves, as well as by other smaller companies such as Internet cafés, which receive a percentage of the benefits for their contribution in storage space. Unlike Ivy, Oceanstore is not a Unix-like file system. Instead, Oceanstore provides a proprietary interface that allows applications to insert, lookup, and update data objects using a wide range of possible semantics. A primary tier of nodes using Byzantine-fault tolerant replication serializes all updates and propagates them to the rest of the network.

The Cooperative File System [17] is a read-only file system built upon the DHash distributed hash table and Chord routing layer [16]. Since its file system structures are very similar to that of SFSRO [11], it suffers from the same problem: each time a file is modified the file system owner must reinsert all CHBs from that file up to the root block, which must also be resigned and reinserted.

Farsite [9] is a decentralized distributed file system that uses untrusted computers to store file system data. Although it has several of the characteristics of peer-to-peer systems (decentralization, each node behaving as client and server, etc.), Farsite is designed to be deployed in a local area network, typically that of a large corporation or university. Therefore, network topology is ignored and a scale no larger than 100.000 nodes is assumed. No key-based routing algorithm is employed here. Instead, the IP addresses of the machines that store the replicas of a given file are included within the file's metadata. Since Farsite is designed to emulate a local NTFS file system, its designers paid special attention to the security and consistency aspects of the system, thus implementing a complex trust and lease scheme.

Pangaea [19] is a wide-area peer-to-peer file system designed at HP Labs. Rather than using a KBR layer, Pangaea's particular design keeps a graph of the nodes on which the replicas of a given file or directory are stored. This graph is then used to disseminate updates and to discover new nodes when a replica must be added. In order to maximize access speed, when a nodes accesses a file it creates a local replica of that file and adds itself to the replica graph. Although the cost of adding and remove a replica is constant, that of propagating updates is proportional to the number of replicas, as this is done by flooding the replica graph. Namespace conflicts in Pangaea are automatically solved using a combination of techniques known as backpointers, namespace containment, and directory resurrection.

7 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 peer-to-peer systems are particularly subject to network latencies.

Another equally important factor is the choice of the consistency model. A strict consistency guarantee can impair performance significantly. Therefore, a large-scale 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. Future work will involve envisaging and adding new consistency models to the prototype.

Ongoing work focuses on developing a more complex conflict detection and resolution scheme, such as one based on version vectors. We also plan to provide support for concurrency control primitives, such as exclusive file creation. This will support applications that use file locks for concurrency control.

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] 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.

[2] 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.

[3] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. of IPTPS*, 2003.

[4] I. Clarke, O. Sandberg, B.Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46-66, July 2000.

[5] L. Rizzo. Dummynet and Forward Error Correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, June 1998.

[6] FreePastry. `http://freepastry.rice.edu/`

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] K. Fu, M. Frans Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181-196, October 2000.

[12] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. In *ACM Computing Surveys*, 22(4):321-375, Dec. 1990.

[13] 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).

[14] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation* (OSDI 1999).

[15] D. Mazières. A toolkit for user-level file systems. In *Proc. of the Usenix Technical Conference*, pages 261-274, June 2001.

[16] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, August 2001.

[17] 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.

[18] 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.

[19] 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.

Contents

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399