Exploiting Network Locality in a Decentralized Read-write Peer-to-peer File System

Fabio Picconi, Jean-Michel Busca, Pierre Sens
LIP6, Université Paris 6 - CNRS, Paris, France
INRIA, Rocquencourt, France
{fabio.picconi, jean-michel.busca, pierre.sens}@lip6.fr

Abstract

We have developed a completely decentralized multi-user read-write peer-to-peer file system with good locality properties. In our system all data is contained in blocks stored using the Past distributed hash table (DHT), thus taking advantage of the fault tolerance and locality properties of Past and Pastry. We have also introduced a modification to the Past DHT which allows us to further increase performance when using a relaxed but nevertheless useful consistency model. Authentication and integrity are assured using standard cryptographic mechanisms.

We have developed a prototype in order to evaluate the performance of our design. Our prototype is programmed in Java and uses the FreePastry open-source implementation of Past and Pastry. It allows applications to choose between two degrees of consistency. Preliminary results obtained through simulation suggest that our system is approximately twice as slow as 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 [3, 7, 8, 10, 14, 16], 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 is still very limited.

One of the reasons for this is that maintaining consistency, ensuring security, and achieving good performance as the system grows to a very large scale is not an easy task. Read-only systems, such as CFS [14], 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, and simplifies the security model since only the administrator can issue updates to 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 [10], 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. However, since each Ivy user has its own log, increasing the number of users sharing a given file implies traversing a larger number of logs when accessing a file. Another limitation in Ivy is that applications have little control on data consistency. Although Ivy uses a consistency model similar to close-to-open consistency, applications cannot fully decide when written data is propagated to the network.

Oceanstore [8] uses a different approach to handling updates by introducing some degree of centralization. A primary tier of nodes uses a Byzantine-fault tolerant (BFT) [11] 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 also takes into account network locality to optimize replica location. Locality management is completely absent in Ivy.

We have designed a highly-scalable, completely decentralized multi-writer peer-to-peer file system. For every file or directory our system 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 achieved by signing inodes before inserting them into the Past network. For each inode the system generates a public-private key pair which is used by clients to sign the Public-Key Block (PKB) in which the inode is stored. Data blocks are stored in immutable Content-Hash Blocks (CHB) so that their integrity can be easily verified. PKBs and CHBs are well described in [10]. All blocks are replicated in order to improve fault tolerance and to reduce the impact of network latency.

We have implemented a prototype written in Java. It runs on a modified version of the FreePastry [4] 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 shows how a modification to Past's DHT interface can be used to increase efficiency in our file system. It describes a consistency model which lies between close-to-open and the read-your-writes guarantee, and can be efficiently implemented using our modified Past service.

The remaining part of this paper is as follows: section 2 briefly 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 modified 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. Pastry and Past

Pastry [2] is a self-organizing, fault-tolerant, decentralised 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) which is randomly assigned. The nodeid space can be thought of as a circle ranging from 0 to $2^{idlen} - 1$ (see Figure 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 fixed-length digits. When a message is routed, each hop forwards the message to a node whose nodeid shares a larger prefix with the routing key, or is numerically closer to the key. Figure 1 shows an example of a message with key $d46a1c$ being routed from node $65a1fc$.

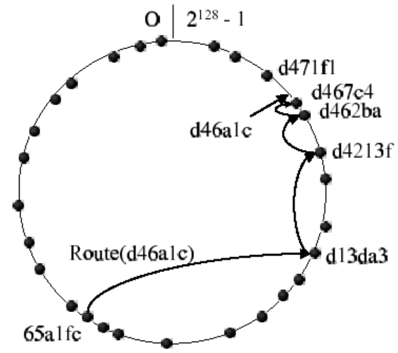

Figure 1. Pastry message routing

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 proximity metric. Messages are therefore routed following an locality-optimized path. In addition, Pastry's routing algorithm is highly efficient. If the routing tables are accurate, 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 $N = 106$ and $b = 4$, this expected hop count is 5.

Past [6] is a highly-scalable peer-to-peer storage service. It 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 stored in the Past nodes whose nodeids are closest to the object's key, and therefore adjacent in the nodeid space. This ensures that at least one replica is always reachable provided all replicas have not failed simultaneously, a very improbable scenario since nodes with adjacent nodeids should exhibit a very low fault correlation (Pastry nodeids are randomly assigned throughout the network).

Finally, Past benefits from Pastry's locality properties when placing and retrieving replicas. In particular, when a Past client retrieves an object replica, it is highly likely that the found replica will be the one closest to the client. As we will see in a following section, this property helps considerably in achieving good performance when using a relaxed consistency model.

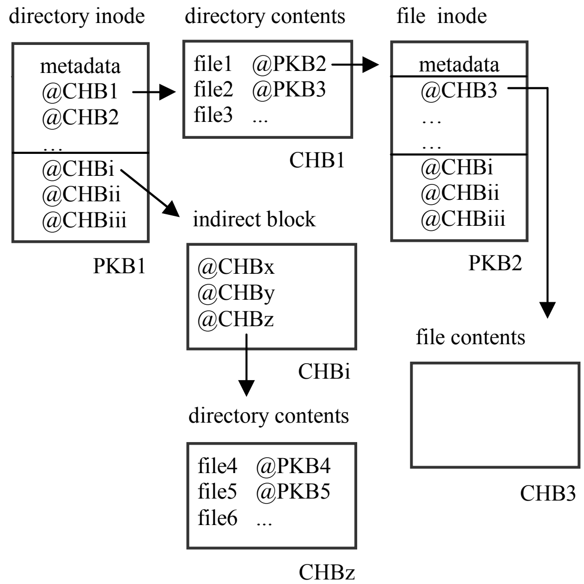

Figure 2. file system structure

3. Design

We begin the description of our file system by presenting how file system data is stored on the network. The data structures used in our design resemble 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 2, each inode is stored in a Past Public-Key block (PKB). The corresponding private-public key pair is generated when the file described by this inode is created, and is stored encrypted within the inode itself. 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. 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, which consist basically of a file name and the Past address of the corresponding inode.

3.1. Updates and conflicts

As we mentioned in the previous section, our file system structures are similar to those of CFS [14]. 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 he knows the root block's private key. In our design, we choose to use PKBs to store inodes, thus eliminating the cascade effect of CFS's inode modification.

Modifying a file or directory in our system requires updating the PKB in which its inode is stored, but it also usually involves the insertion of new CHBs. For instance, in order to update a file the file system client must perform the following operations:

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.*
5. *Update the inode's block pointer table with the address(es) of the new block(s).*
6. *Update the PKB 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 blocks 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 [10], 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. Each time a client generates a new version of an inode, it timestamps the inode using the client's local clock. Then, when the inode is inserted into the Past service, its timestamp is compared to that of the existing inode. If the latter is newer than the newly inserted inode, the block is not overwritten. It should be noticed that this mechanism requires that clients loosely synchronize their clocks using NTP or some other clock synchronization protocol. A similar mechanism is employed by the Pangaea system [16].

Given the limitations of this model 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. The design and evaluation of further consistency models is the subject of current research.

Close-to-open consistency [5, 9] is a relaxed consistency model which was employed in the original AFS distributed file system. It has been partially implemented in some NFS systems. 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 our system, close-to-open consistency is implemented by getting the latest inode from the network (by retrieving all replicas) when the file is opened and keeping a cached copy until the file is closed. New blocks are also buffered until the file is closed in order to avoid network latencies.

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 second consistency model that we have implemented is weaker than close-to-open consistency, and can be useful for applications which access files that are seldom shared. It is based on the *read-your-writes* session guarantee originally introduced by the Bayou system [15]. Whereas the close-to-open model requires that the latest version of a file be retrieved from the network upon a file open, the read-your-writes model only requires that the client retrieve a version that reflects all previous modifications issued by that client. The retrieved version may not be the latest in the network, but may still be valid for that client. Since the consistency constraint is weaker, the model allows for more agressive caching and less network accesses, thus improving performance.

In our version of the read-your-writes model, we ensure that read operations always reflect all previous local writes, even if the file is closed and later reopened. However, our model provides better consistency than the original Bayou guarantee by making highly it likely that a file open will reflect any distant writes propagated after a file has been closed. In 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 model uses a two-phase procedure. When a file is opened, the file system client first instructs Past to retrieve a single replica of the inode, an operation which, given Past's locality properties, is relatively cheap in network terms. If the retrieved replica reflects the client's previous accesses (according to its timestamp), then no other network accesses are necessary. Otherwise, a second phase is executed in which the client retrieves all inode replicas and keeps that with the most recent timestamp, which will naturally reflect previous accesses. 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, two phases are involved when retrieving an inode replica with the enhanced read-your-writes guarantee. 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 (i.e., some metadata) of all live replicas, and then lets the application decide which replica is to be retrieved.

The modification we introduced allows our file system to specify a constraint over the block's metadata when performing a *lookup* call. In this way, if the first replica encountered by the Past lookup message 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 must execute the second phase 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.

3.4. Security

As we pointed out above, every inode in the file system is stored in a different Public-Key Block (PKB). In order for PKBs to be stored, the block's contents must be signed with the PKB's private key. The block is then inserted into the DHT using the hash of its private key as the DHT key.

In this way, both the DHT service and the clients can verify the block's integrity by checking that the signature is valid.

The problem that arises is that of securely storing the private keys of all PKBs in the system. Our solution consists in encrypting a PKB's private key using a symmetric cipher, and storing the encrypted PKB's private key within the PKB itself. We can imagine choosing a symmetric key for a given group of users having write access to a series of files, and distributing the symmetric key only to this groups of users. Users having read-only access do not need the symmetric key to access the filesystem.

Our design also allows for protecting against Byzantine faults. Past nodes exhibiting arbitrary behaviour may deny the existence of a previously stored block, or return an old version upon reception of a *lookup* message (rollback attack), but cannot compromise the integrity of a PKB, which is digitally signed. However, a file system client will retrieve all inode replicas when using the close-to-open model, or when the replica retrieved in the first phase of the read-your-writes model is not valid. This guarantees that the client will retrieve a replica which is valid according to the chosen consistency model provided at least one replica is not faulty.

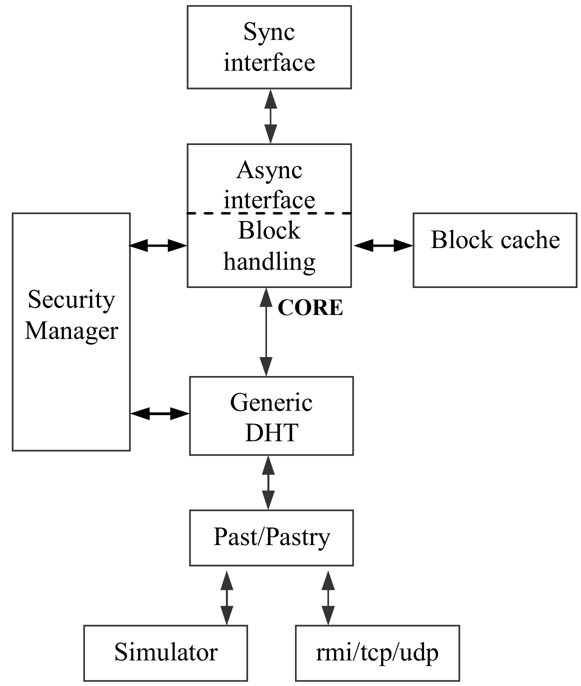

Figure 3. prototype architecture

4. Prototype

Our prototype is written entirely in Java 1.4, and thus runs on any platform that supports the JVM 1.4. Figure 3 shows a diagram of the different software components. All blocks are coded in Java and are executed within the same Java VM. At the bottom of the stack, the Pastry service relies on either the standard Java RMI/TCP/UDP transports, or a modified version of the network simulator included in the FreePastry 1.3 release.

4.1. Simulator

Our simulator is based on that provided by FreePastry 1.3, which we have modified in order to introduce network latency between two nodes. In our modified version, when a Pastry node sends a message to another node the simulator calculates the network latency for the message according to a specific network model, and delays the processing corresponding to the reception of the message by that calculated amount of time.

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 file system layer code (i.e. 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 [5]. 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.

In all our tests we run a single instance of the Andrew Benchmark program. 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. All simulations are run on a Pentium 4 2.4 GHz with 512 Mbytes of RAM.

5.1. Consistency models

In this test we measured the impact of the choice of the consistency model on the expected performance. We simulated a network of 4096 nodes using a inter-node delay of 50 milliseconds. Past was configured to generate 16 replicas of each inserted object. The directory used for the benchmark contains some source and header files, for a total of 55 files and 450 Kbytes, and does not contain any subdirectories. The client data cache is set by default to 2 Mbytes.

We performed three test runs. In the first we used the close-to-open (CTO) consistency model. In the second we

Phase	CTO	RYW	RYW*
1	-	-	-
2	72,2	71,1	71,5
3	22,8	9,2	9,9
4	23,2	9,1	9,9
5	328,1	164,5	168,5
Total	**446,3**	**253,9**	**259,8**

Figure 4. execution time (in sec.) for both consistency models. *in this run 10% of the updates are discarded.

activated only the read-your-writes (RYW) session guarantee. In the third run we used the same model as in the second, but we created some stale inode replicas by preventing one inode replica out of ten from being updated. This method simulates an execution in which one tenth of an inode's replicas would be down when inodes are updated in phase 2. These replicas would be up again when the file system client must retrieve an inode replica in phases 3-5.

As expected, Figure 4 shows that close-to-open consistency is much more expensive than the read-your-writes session guarantee. In fact, whereas the former requires that all replicas be retrieved to ensure that an open reflect all distant writes, the latter need only find a replica newer than that last seen at the local client. The results also show that the presence of stale inode replicas degrades performance since some Past *lookup* calls fail, forcing the client to execute the slower second phase. However, performance degradation over the total run-time is very small, about 2%.

5.2. Replication factor

The performance obtained when using the read-your-writes model depends on how close the retrieved replica is to the client. Increasing k, the replication factor, increases the chance that a close replica will be found, but generates more traffic and consumes more storage space. We ran the Andrew Benchmark with different replication factors to determine its impact in execution time when using the read-your-writes model. In this test we used a simulated network of 1024 nodes. In the first run we disable replication altogether, and then we increase k up to the value 11.

First we measured the execution time of phases 2 (file write) and 4 (file read) of the Andrew Benchmark as a function of the replication factor. We simulate variable inter-node latencies by using the "Sphere" topology. Each Pastry node is located at a given point on the surface of a sphere, and the network latency between two nodes is proportional to the Euclidean distance between the two points of the surface of the sphere. We use a maximum latency

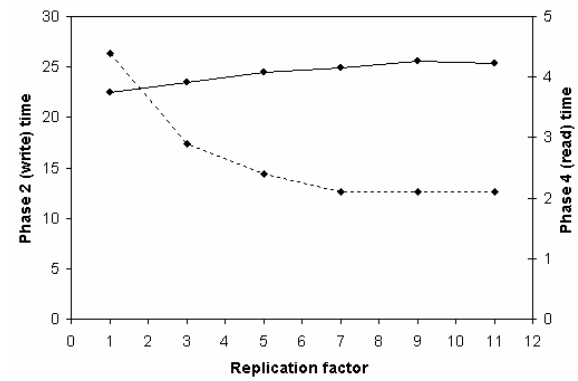

Figure 5. execution time (in sec.) of the read (dashed) and write (solid) phases of the Andrew Benchmark using the Sphere topology

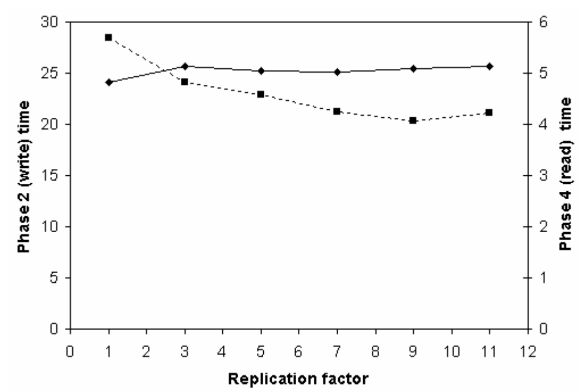

Figure 6. idem as Figure 5, using a constanct delay and distance (according to the proximity metric) between nodes

of 100 ms., which corresponds to two diametrically opposed points. As shown in Figure 5, read time improves by aprox. 100% when increasing k from 1 to 7 and stabilizes thereafter. Although replicas are inserted in parallel, we observe that write time increases by 10% between $k = 1$ and $k = 11$. Using more replicas clearly increases the probability that one of the replicas be distant from the client, i.e. that longer latencies be involved when updating it.

Next, we repeated the same test, this time using a constant latency and Pastry distance between nodes. In this case our system does not benefit from Pastry's locality properties. The results, depicted in Figure 6, show that read performance improves only by 30%, compared to the 100% obtained previously. Increasing the number of replicas makes

Phase	NFS	Our system
1	-	-
2	19,6	39,6
3	3,5	3,9
4	9,9	4,2
5	53,6	75,4
Total	**86,6**	**123,1**

Figure 7. execution time vs. NFS (in sec.)

more likely that a close replica will be found, but the performance gain is less than when using locality-optimized routing tables. This confirms that using a DHT service with good locality properties considerably lowers our system's access time during read operations.

5.3. NFS comparison

Our last measurements were aimed at comparing our system to NFS. In this case we use a simulated Past network of 128 nodes and a 25 millisecond inter-node delay. Past is configured to use a replication factor of 16. The input directory is the same as in both previous configurations. We compare this configuration with a client accessing a single NFS server whose RPC responses are received 50 milliseconds after the corresponding request is sent. We simulate this delay using a modified version of the dumbfs program included in the SFS Toolkit [12].

As show in Figure 7, our simulated execution is slightly less than 50% slower than NFS for a network of 128 nodes. This difference can be expected to grow as the number of nodes increases (given the greater average number of routing hops). Other measurements not presented in this paper suggest that increasing the number of nodes from 128 to 4096 increases total run-time by 40%, in which case our system would be approximately twice as slow (100%) as NFS. This would suggest that our system yields better performance than Ivy and Oceanstore, which are between two to three times slower than NFS.

6. 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.

Our design is not finished. 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 suggests that performance is only half of that of NFS. However, our results are still preliminary and must be corroborated by further evaluations using more realistic test conditions.

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] 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.

[4] FreePastry. `http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry/`

[5] 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.

[6] 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*, October 2001.

[7] 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.

[8] 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.

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

[10] 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).

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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)*, Boston, MA, USA, December 2002.