Using Transparent Files in a Fault Tolerant Distributed File System

Marcelo Madruga, Sergio Loest, Carlos Maziero
Graduate Program in Computer Science
Pontifical Catholic University of Paraná
Curitiba, Brazil
E-mail: {mcmadruga,sloest,maziero}@ppgia.pucpr.br

***Abstract*—The peer-to-peer model and the bandwidth availability are fostering the creation of new distributed file systems. However, files belonging to local application and distributed applications are usually handled in the same way by the local file system, so both contribute equally to consume storage space. This paper presents a peer-to-peer distributed file system which uses the "transparent file" concept to improve its fault tolerance and file availability. Files are kept as transparent/volatile replicas, using the free space available in each local file system. When a replica is invalidated, peers cooperate to restore it. The proposed architecture was implemented and tested; experiments showed its feasilibity, and that its costs are proportional to the size of files being replicated. The occurrence of multiple simultaneous replica invalidations did not impose a significant overhead.**

I. Introduction

With the growth of computers' interconnections, Internet, and available bandwidth [1], distributed file and storage systems are moving from a client/server model to a peer-to-peer model [2], [3]. Peer-to-peer systems operate in a decentralized manner, having the ability to adapt themselves to failures through self-organization, with a small impact on performance and connectivity [4]. Due to such characteristics, distributed file and storage systems can benefit from this architecture, by providing fault tolerance, higher amount of available storage, and higher availability to users. Several new peer-to-peer storage systems have been proposed recently [5], [6], [7].

Traditionally, files belonging to distributed applications and files belonging to local applications are handled in the same manner by the local file system. Thereof both types of files contribute in the same way to consume available storage space. However, this lack of distinction brings several concerns and problems when deploying distributed file systems in a peer-to-peer fashion. There are some peer-to-peer storage systems that may be usually seen as contributory applications, where users donate free disk space, but the donated resources are not directly consumed by the donor. In that case, users' behavior is completely different from each other. Some are worried when donating big amounts of free disk space, because they will eventually need the contributed space; although others are willing to donate hundreds of gigabytes of free space because they can access more content [8], [9].

Several ideas to solve or diminish the problems caused by such user's behavior have been recently presented [10], [11]. One of them is the local *Transparent File System* (TFS), presented in [9]. TFS is a local file system that can contribute 100% of the free disk space while imposing a minimum performance impact on the local file system and local applications. TFS separates files into "opaque files" and "transparent files". Transparent files are stored in the free storage area, but their space can be reclaimed at any time by the local system. Therefore, "transparent files" don't appear to local users as storage consumers, giving them the impression that no free space is being contributed. "Opaque files" are the usual local files stored in the disk.

In this paper, we propose a peer-to-peer file system offering high data availability and fault tolerance through the use of transparent files, to maximize the overall storage space donated for the system. Transparent files impose a new challenge, because they may be removed from local systems at any time and unpredictably. Our design handles such removals in a graceful way, adjusting itself to keep the high availability of the files. This paper is structured as follows: Section II describes the current *p2p* file system design; Section III presents the Transparent File System approach; Section IV discusses our system's architecture and the algorithms used to handle transparent files deletions properly; Section V describes the prototype and the preliminary evaluation results; finally, in Section VI the related work is discussed and in Section VII some conclusions are drawn.

II. Peer-to-Peer Storage Systems

Initially, Peer-to-peer (*p2p*) systems were created to anonymously share files on the Internet, but other domain areas became interested in *p2p* properties (e.g. flexibility, inherent scalability, and smaller costs). Today, there are *p2p* systems for audio and video streaming, instant messaging, and distributed file and storage systems [2] [4]. In the *p2p* model, participants (*peers*) act as clients and servers of the service being provided. A *Distributed Hash Table* (DHT) is a *p2p* service that stores [*key,value*] pairs. It also allows the reliable and efficient insertion and retrieval of information in a overlay network with a large amount of peers. Typical DHT implementations include *Chord* [12] and *Pastry* [13].

The *Cooperative File System* (CFS) [7] is a read-only distributed file system, where only the file owner has the right to modify its contents, but others can read the files. CFS uses the *Chord* DHT service to store and lookup files' metadata.

All peers use the same algorithm to transform the metadata present in the DHT into a file system-like abstraction, with file and folders hierarchy. CFS uses replication to maintain high data availability. Files inserted in the system are divided into several data blocks that are replicated to k servers. CFS tries to keep k replicas of each block in the system, migrating blocks when nodes join or leave.

Ivy [14] is multi-user, read-write, and log-based distributed file system, created on top of the *Chord* substract. Each peer participating in the file system has a log to store all file modifications and additions. These logs are stored in the DHT provided by *Chord*. Each peer can read and look up information in all logs stored in the DHT, but can only write in its own log. Ivy offers a session consistency model, in which updates become visible only after the file is closed.

FARSITE [5] is a serverless distributed file system that runs in untrusted computers, that uses techniques of fault tolerance to provide data high availability: file replication and scattering, data and communication cryptography and Byzantine-fault tolerant commit protocols. Its goal is that a set of desktop client computers can collaborate to establish a virtual file server that can be accessed by any client at any time. Thus, the FARSITE operates as a unique central file server.

PAST [6] is large-scale, peer-to-peer archiving storage utility that provides scalability, availability, security, and it is build on top of the *Pastry* peer-to-peer substrate. PAST offers a unique and transparent name space, because each files stored in the system has a unique identifier (*fileid*). One of PAST's characteristic is the ability to maintain the number of replicas invariant (*replication factor*), thus faulty nodes and network partition are tolerated without loss of data availability.

III. The Transparent File System

The *Transparent File System*(TFS) [9] is a local file system that introduces the concept of "transparent files". A transparent file is a file that is being stored by the local file system, but it is not visible (or perceived) by the local users. From the local users' point of view, no space is being donated at all. In opposition, normal files belonging to the local users are called "opaque files", and are all locally visible.

TFS's core is its block allocation scheme, which does not guarantee the persistence of transparent files. Opaque files have precedence over transparent ones: transparent files are stored in the free disk space, but may be replaced and overwritten at any moment with opaque files by the local system. TFS was implement as a modification of the Linux Ext2 file system, and adds three new states to its standard block allocation scheme: *Transparent*, *Allocated-and-Overwritten*, and *Free-and-Overwritten*. The *Transparent* state indicates that the block is being used by a transparent file. The *Allocated-and-Overwritten* and *Free-and-Overwritten* states signal that the block was being used by a transparent file and was overwritten by an opaque file, and, later was freed, respectively.

Transparent Files provides an interesting basis to build a distributed file system to provide data redundancy and fault tolerance: files can be replicated over several nodes transparently, using their free disk space. However, as local applications may request storage space for opaque files, transparently stored replicas may be overwritten or deleted at any time, unpredictably. The current TFS implementation does not provide any mechanism to notify about transparent files deallocation; this problem can only be detected when a process tries to open the deleted transparent file.

IV. A Distributed File System based on Transparent Files

In this section, we propose a fault-tolerant distributed file system that uses the benefits of the Transparent File System and circumvents the obstacle of the unpredictably deletion of transparent files. In our proposal, a group of nodes share their free storage space via TFS, and where replicas of files are stored as transparent files. Each node has a local peer that manages the locally stored replicas, interacts with other peers to maintain the replication scheme, and offers replicated files to applications. Our design is targeted to applications that provide high availability by keeping an invariant number of replicas for any given file, similarly to PAST and CFS. Thus, files are accessible even in the presence of network partitions, faulty nodes and transparent files deletions. Other distributed file systems' aspects, such as data consistency semantic, were not taken in consideration because the scope of this paper is a viability study of transparent files utilization.

We introduce the concept of *replica invalidation*, which indicates that a replica of a transparent file has been deleted or overwritten. Whenever such invalidation happens, the system has the opportunity to re-arrange the replicas and re-establish the appropriate number of replicas, ensuring high availability.

A. Architecture

The proposed architecture consists of a set of nodes connected by a p2p overlay network. All peers are equal, work in a cooperative way and are organized in an unstructured network. Each node is responsible to manage its transparent storage, to detect possible transparent file deletions, to download or upload replicas from/to other nodes and to provide an interface to local users. Each peer has seven components: a *Distributed File application*, a *Distributed File System*, a *Storage Manager*, a *Replication Manager*, a *Download Manager*, a *Local Transparent File System* and a *Peer-to-Peer Substrate*, depicted in Figure 1 and explained hereafter.

The *Distributed File Application* uses the distributed file system to provide a service to users, such as a digital media library. The *Distributed File System* is responsible for offering a file system-like interface and for controlling the system's overall behavior. It controls the *Storage Manager* and the *Replication Manager* to store and retrieve files through the local storage or the network.

The *Storage Manager* handles requests to store and retrieve transparent files from the local disk. It is also responsible for watching the local transparent file system to detect transparent file deletions. If a transparent file is deleted, the Storage Manager notifies the Replication Manager immediately. The

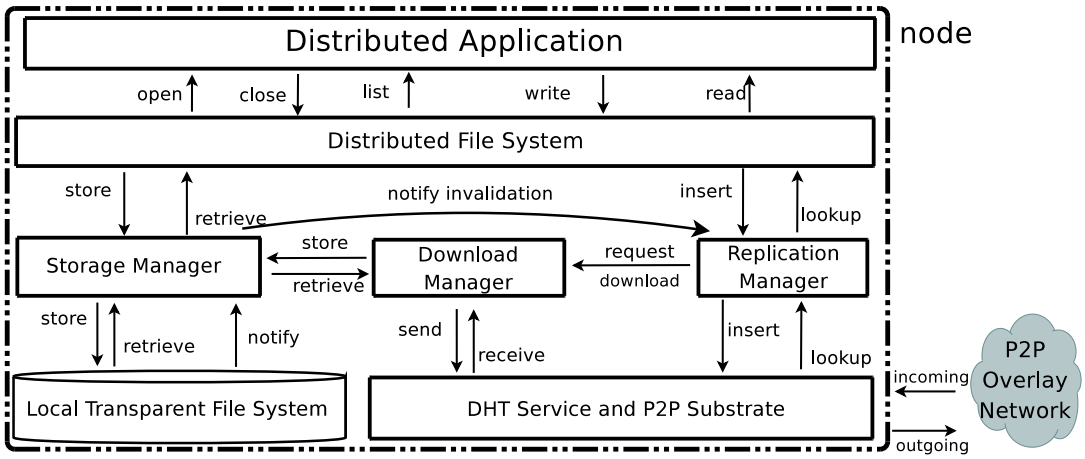

Fig. 1. Architecture model

Replication Manager keeps the number of replicas of a file constant in the system. Periodically, it verifies the number of replicas of each file it owns; if any insufficiency is detected, it starts a replication procedure to re-establish the correct number of replicas. The Replication Manager also executes the replica invalidation recovery procedure, as explained in Section IV-D.

The *Download Manager* sends and receives files replicas over the p2p overlay network. It is triggered when a file to be opened is not in the local transparent file system and should be gotten from another node. The Replication Manager invokes it also, when replicas need to be transferred. The *local TFS* is responsible for contributing the free disk space and for storing all replicas as transparent files. Finally, the *Peer-to-Peer Substrate* is responsible to connect the nodes in a overlay network, maintain the connectivity among nodes, manage node membership through join/leave protocols, and route messages.

B. Basic PAST/Pastry Concepts

As our proposal is based on PAST and Pastry *p2p* substrate, it is important to define some of their concepts and services offered to applications. Pastry [13] is an overlay network message routing and communication service. Each Pastry node has a unique identifier (*nodeId*), a routing table and maintains local information about two sets of nodes: the *Neighborhood* set $\mathbb{N}$ (the set of the closest nodes relatively to a proximity metric), and the *Leaf* set $\mathbb{L}$ (set of nodes whose IDs are numerically closest to that node).

PAST [6] is a storage service built on top of Pastry, which allows to store and retrieve files in several nodes with replication. A replication factor k can be defined for all stored files. It includes a Distributed Hash Table (DHT) service which provides an interface offering *put(k,v)* and *get(k,v)* operations.

C. Handling File Access

In order to insert a new file in the system, the Distributed File Application uses the interface provided by the Distributed File System layer, to inform the file name and the file location. The Distributed File System then computes a file identifier *fileid* and obtains the file's metadata. In possession of the *fileid*, its metadata and its location, the Replication Manager identifies which nodes shall hold a replica of the file, based on the PAST/Pastry replication factor k and the numerically closest node identifiers in relation to the *fileid*. Then, the metadata is stored in the DHT, and the selected nodes are informed that they should hold a replica of that file. Each selected node requests the replica's content to the node where the file was inserted, through its Download Manager. After the replica transfer finishes, it is locally stored as a transparent file.

When the Distributed File Application wants to access an existing file, the Distributed File System first checks if a local replica of that file exists. Otherwise, the Replication Manager retrieves the file's metadata from the DHT, instructs the Download Manager to retrieve a file replica from another node and stores it locally as a transparent file. Once the download is complete, the application request can be fulfilled.

D. Handling Replica Invalidation

TFS's trade off for low performance impact and less psychological effect over data persistence is the key aspect on its design, enabling the local file system to donate all free disk space to applications [9]. However, this property leads to a problem when using the TFS to build a distributed file system. In addition to node faults, now the system must now tolerate data persistence faults. To cope with this, we propose a replica invalidation recovery procedure that enables the distributed file system to keep operating even during the presence of simultaneous deletions or overrides of transparent replicas of files. Hereinafter, any transparent replica deletion is called *replica invalidation*. The recovery procedure is triggered whenever a replica is invalidated in any node, and its goal is to keep the number of available replicas constant in the system.

Our system is composed by P peers $p_1 \dots p_n$. A file seen by the distributed file application is denoted as f, and its size is $size(f)$. A replica of f stored at peer p_i is denoted as $r_i(f)$. The set of peers that hold a replica of f is denoted as $\mathbb{R}(f)$. The neighborhood set of a peer p_i and its leaf set,

as provided by the underlying Pastry middleware, are defined here respectively as $\mathbb{N}(p_i)$ and $\mathbb{L}(p_i)$. Finally, the free storage space available at the peer p_i is denoted $fspace(p_i)$.

Considering a peer p_i, when its Storage Manager detects that a local replica $r_i(f)$ was deleted or overwritten, it sends a replica invalidation notification $repl\_inv(f)$ to its Replication Manager. The Replication Manager then queries the Storage Manager about the free space locally available $fspace(p_i)$. If there is enough space to restore the replica, the Replication Manager queries the DHT to get the *nodeId* of a peer p_j holding a replica of f ($p_j \in \mathbb{R}(f)$)[1], and sends a file request $file\_req(f)$ to it. Otherwise, if there is not enough free space, the Replication Manager queries the current f replica holders set $\mathbb{R}(f)$ and sends a replication request $repl\_req(f)$ to the first peer in its leaf set $\mathbb{L}(p_i)$ not holding a replica of file f, denoted here as p_k. The actions executed by the Replication Manager at peer p_i are summarized in the Procedure 1.

Procedure 1 RM at node p_i receives a replica invalidation notification $repl\_inv(f)$

```
1: if fspace(p_i) ≥ size(f) then
2:    Retrieve p_j ∈ R(f) from DHT
3:    Send file_req(f) message to p_j
4: else
5:    Retrieve R(f) from DHT
6:    Retrieve p_k = first(L(p_i)\R(f)) from DHT
7:    Send repl_req(f) message to peer p_k
8: end if
```

In the sequence, to complete the recovery procedure, the Replication Manager at node p_k executes the Procedure 2 after receiving the $repl\_req(f)$ message from peer p_i. Its first action is to verify if there is a local replica of the file f, by querying its Storage Manager. If p_k does not have a replica of f and if there is free storage space available, a DHT lookup is performed to locate another peer p_m holding a replica of f ($p_m \in \mathbb{R}(f)$). Finally, a file request $file\_req(f)$ is sent to p_m. Hence, node p_k is now responsible for a replica of file f, instead of node p_i.

In the other hand, if p_k does not have enough free space to hold a replica, the replication request $repl\_req(f)$ received from p_i is forwarded to the next node in its leaf set $\mathbb{L}(p_k)$ not holding a replica of file f. Otherwise, if node p_k already has a replica of f, it means that multiple replica invalidations of f happened simultaneously, and node p_k was already selected during the execution of procedure 1 by another node. Thus, the replication request $repl\_req(f)$ should also be forwarded, in order to maintain the number of available replicas.

V. Implementation and Evaluation

We developed a prototype of our proposal to verify the usability of transparent files in a distributed file system, and to check if our replica invalidation procedure is able to maintain the file availability. This prototype can be seen as a proof-of-concept and has a very minimal feature set that enables an evaluation under stable conditions, i.e., in the presence of persistence faults only.

[1]In fact, due to the Pastry routing algorithms, usually the node $p_j \in \mathbb{L}(p_i)$ with the smaller number of routing steps will be informed.

Procedure 2 Peer p_k receives a replication request $repl\_req(f)$ from node p_i

```
1: if ∄r_k(f) then
2:    if fspace(p_k) ≥ size(f) then
3:       Retrieve p_m ∈ R(f) from DHT
4:       Send file_req(f) message to p_m
5:    else
6:       Retrieve p_m = first(L(p_k)\R(f)) from DHT
7:       Forward repl_req(f) message to p_m
8:    end if
9: else
10:   Retrieve p_m = first(L(p_k)\R(f)) from DHT
11:   Forward repl_req(f) message to p_m
12: end if
```

A. Prototype Implementation

The prototype was implemented using four open source projects: FreePastry [15], INotify [16], JNotify [17], and the TFS implementation [9]. They were adapted and glued together by around 2,500 lines of Java code. *FreePastry* is a open source implementation of the *Pastry* peer-to-peer substrate and PAST storage utility, written in Java. *INotify* is a Linux kernel module that provides a mechanism to receive event notifications from the file system, indicating if a file or directory has been deleted, renamed, created or modified. *JNotify* provides Java bindings for the Linux *INotify* API, allowing Java-based applications to monitor file system events.

The Java code is responsible for connecting all these components and offers the base to create an application on top of it. It is composed of four modules according to the architecture presented in the previous section: *Simple File Storage Application*, *Storage Manager*, *Replication Manager*, and *Download Manager*.

The Simple File Storage Application relies on services provided by PAST to implement a flat file system abstraction. When a file is inserted in the system, the first action is to calculate its *fileid*, a quasi-unique 168-bit value based on a cryptographic hash of the file name, the local node identifier and a random number. The application then reads the file's metadata, to get the file information, such as size and creation date. The file metadata is then inserted in the PAST and a replication factor k is assigned to it. The file metadata is stored in the DHT and routed to all nodes that shall hold a file replica. After a node receives the new file metadata, stores it in the local file system as an opaque file, and retrieves the file's contents, which is stored as a transparent file. In addition, both procedures described in section IV-D were implemented.

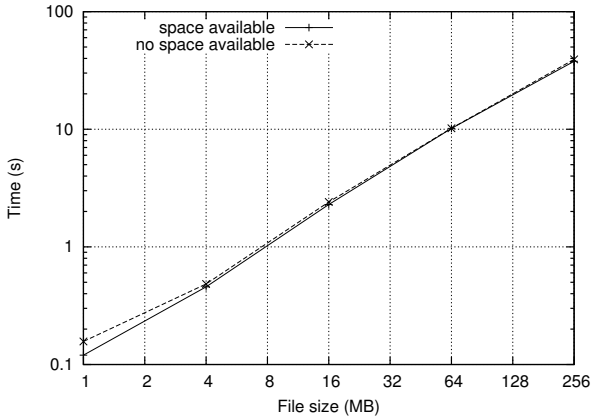

Fig. 2. Recovery time for one replica invalidation

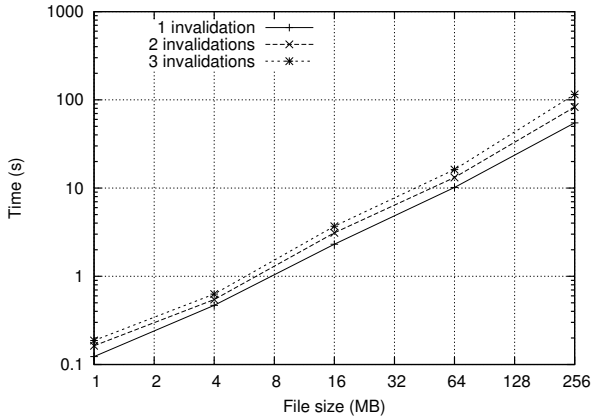

Fig. 3. Recovery time for multiple simultaneous replica invalidations

B. Prototype Evaluation

All experiments were executed in a single machine, a HP AMD Turion 64 X2 with 2.1 Ghz and 3 GB RAM, running Linux Debian Sarge 3.1, Sun's Java SE Runtime Environment version 1.6.06, and the TFS Linux kernel module.

The system was brought up with 20 independent nodes to form the p2p overlay network. A virtual network interface was defined by each node to create socket connections and transfer files through the overlay network. This virtual network interface has a bandwidth throttling mechanism to simulate different bandwidth conditions. After all nodes are up and running, the distributed file system is populated with 25 sample files with their sizes varying from 1 MB to 256 MB, each one having the same replication factor ($k = 4$).

Three preliminary experiments were performed, to investigate the prototype behavior in the presence of replica invalidations. All values presented in the charts represent a mean value from three executions of each experiment and the variation was below 5%. In order to make the experiments more easily observable, we deliberately deleted some transparent files instead of forcing TFS to override them.

The first experiment was executed to estimate the overhead caused by the replica invalidation recovery procedure. The time needed to restore a deleted replica is measured in two situations: (1) when the node where the replica was deleted has still free space enough to restore a replica from another node, and (2) when the node does not have enough free space, so it has to find another node to restore the replica. Figure 2 shows the result of this experiment. It can be observed that the overhead caused by the recovery procedure and algorithm is low in comparison with the time needed to transfer the file. Only when small files are invalidated, an overhead is noted.

The second experiment verified the system behavior in a condition of multiple simultaneous replica invalidations (1, 2, or 3 simultaneous invalidations from replicas of the same file). The system was designed to keep a constant number of replicas available. Therefore, it is expected that after some time all invalidated replicas are restored. To exercise the complete recovery procedure and to trigger the system's self-organization, we simulated that each node does not have enough free space to request a new replica copy when its replica is invalidated. Figure 3 shows the times needed to restore all replicas. The system was able to restore all invalidated replicas, and the elapsed times are proportional to the replica's size.

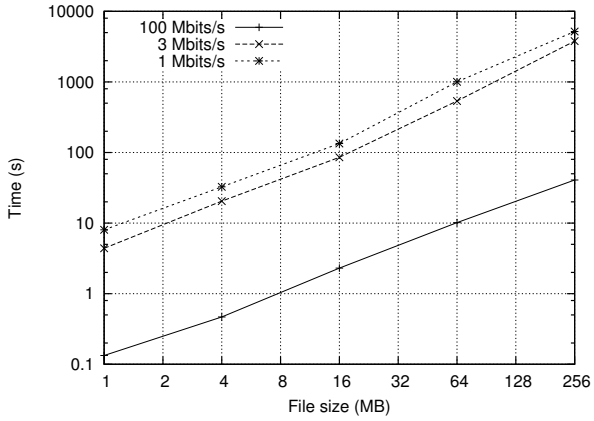

Fig. 4. Recovery time for one replica invalidation for different bandwidths

Finally, the third experiment investigated the system under different bandwidth limitations. Three bandwidth limits were chosen to simulate a local area network and average broadband Internet connections: 100 Mbits/s, 3 Mbits/s and 1 Mbits/s. As depicted in Figure 4, the system keeps its behavior for the three cases. The elapsed time to restore one invalidated replica is proportional to the replica's size and the available bandwidth.

VI. Related Work

TFS introduced the "transparent file" concept quite recently [9], and therefore no similar work, using transparent files to create a distributed file and storage system, was found. However, the transparent contribution of idle resources have been used in different computer science areas for a while. Systems like those proposed in [18] and [19] implement the transparent contribution of idle processing power (CPU cycles) of home computers. Such systems are used to harvest processing power to help solving large and complex scientific

calculations that would be unfeasible in centralized systems, like simulations of weather models and chemical reactions.

The contribution of idle memory is proposed in [20] [21] to improve the performance of disk intensive applications. Free memory is used to cache file's data, so applications can read the contents of a file directly from the memory because the system already copied the data from the disk into the memory. Whenever the system needs more memory to its applications, the cached data is moved back to the disk or purged. In [22], the memory usage is controlled by the operating system in accordance to applications priority. Contributory applications have lower priority to access the memory than local applications. Therefore, the miss rate of page faults is reduced for the local applications.

VII. Conclusion

This paper presented and evaluated a proof-of-concept model to use transparent files, introduced by the Transparent File System [9], in a fault tolerant peer-to-peer distributed file system. In our proposal, a file is replicated among several nodes and its replicas are stored as transparent files. However, transparent files can vanish from the disk in an unpredictable way, therefore an initial mechanism to detect such deletions and to recover the invalidated replicas was proposed. Preliminary experiments shown that the our approach is feasible, and that its costs are proportional to the size of files being replicated. The occurrence of multiple simultaneous replica invalidations did not impose a significant overhead. More detailed experiments using a real wide-area distributed environment and under node churn scenarios are planned.

Several applications can benefit from a large amount of free disk space scattered among several nodes. An example could be a digital library for pictures, videos and documents, in which users can publish and look-up for files. A digital library must provide a unique and independent name space for the files: users should always see the same file hierarchy, everywhere. Therefore, each file must have a unique identifier, belong to a category, and be signed by the author to certify its integrity. In addition, data consistency can be achieved across a single write-lock mechanism, because only the author is allowed to modify his files. After the write operation, the file is released from its lock and can be replicated through the system.

As this was the first attempt to use transparent files, some issues remain open to be improved in the future. First, the proposed replication algorithm is simple and can be polished to handle boundary situation, like when all replicas of the same file are deleted simultaneously. For this situation, a threshold could trigger some emergency actions when the replication level of a file reaches a minimum value. A possible action would be to temporarily convert the remaining transparent replicas into opaque files to guarantee their integrity during a replica shortage. The current TFS implementation only supports transparent file invalidations. If just one disk block occupied a transparent file is requested by an opaque file, the entire transparent file is invalidated. TFS authors proposed (but not implemented) mechanisms to support the invalidation of individual blocks [9]. Using such feature, restoring an invalidated file would be much faster, as only the invalidated blocks had to be restored from another replica.

References

[1] D. Payne and P. Woolnough, "Bandwidth drivers for future networks," *IEE Telecommunication Series*, vol. 47, pp. 21–36, 2004.

[2] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Computing Surveys*, vol. 36, no. 4, pp. 335–371, 2004.

[3] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," in *Intl Conference on Information Technology: Coding and Computing (ITCC'05), Volume II*, 2005.

[4] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, "Peer-to-Peer Computing," HP Labs, Tech. Rep. 2002-57, 2002.

[5] 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," *SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.

[6] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *8th IEEE Workshop on Hot Topics in Operating Systems*, 2001.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *18th ACM Symposium on Operating Systems Principles*, 2001.

[8] P. Golle, K. Leyton-Brown, I. Mironov, and M. Lillibridge, "Incentives for sharing in peer-to-peer networks," in *2nd Intl Workshop on Electronic Commerce*, 2001.

[9] J. Cipar, M. D. Corner, and E. D. Berger, "TFS: a transparent file system for contributory storage," in *5th USENIX Conference on File and Storage Technologies*, 2007.

[10] O. Leonard, J. Nieh, E. Zadok, J. Osborn, A. Shater, and C. Wright, "The Design and Implementation of Elastic Quotas: A System for Flexible File System Management," Computer Science, Columbia University, Tech. Rep. CUCS01402, 2002.

[11] T. wan Ngan, D. Wallach, and P. Druschel, "Enforcing fair sharing of peer-to-peer resources," in *2nd Workshop on Peer-to-Peer Systems*, 2003.

[12] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *ACM SIGCOMM*, 2001.

[13] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329–340, 2001.

[14] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *5th Symposium on Oper. Systems Design and Implementation*, 2002.

[15] P. Druschel and A. Rowstron, "FreePastry," February 2001, `http://freepastry.org`.

[16] R. Love, "Kernel korner: Intro to INotify," *Linux Journal*, vol. 2005, no. 139, p. 8, 2005.

[17] O. Yadan, "JNotify," November 2005, `http://jnotify.sourceforge.com`.

[18] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[19] S. Larson, C. Snow, M. Shirts, and V. Pande, "Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology," *Computational Genomics*, vol. Horizon Press, 2002.

[20] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *IEEE Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[21] R. Patterson, G. Gibson, and M. Satyanarayanan, "A status report on research in transparent informed prefetching," *ACM Operating Systems Review*, vol. 27, no. 2, pp. 21–34, 1993.

[22] J. Cipar, M. D. Corner, and E. D. Berger, "Transparent contribution of memory," in *USENIX Annual Technical Conference*, 2006.