Distributed Data Storage with Data Versioning

Lukáš Hejtmánek

Faculty of Informatics,
Masaryk University,
Botanická 68a, 602 00 Brno,
Czech Republic
xhejtman@mail.muni.cz

Abstract. In this paper, we demonstrate the concept of a distributed file system that supports file versioning, file sharing, and endless files based on sliding window approach. It uses a distributed storage substrate that provides immutable block allocations, and a metadata manager, that maps files to data blocks using metadata. While file versioning and read-write sharing are often understood as contradictory, we demonstrate the approach that allows both properties to coexist. We have described the application-level prototype implementation, that we use for verifying this concept.

1 Introduction

The importance of versioning support is increasing for many classes of applications, as demonstrated by ever broader utilization of version control systems like Concurrent Versioning System (CVS) [3], Subversion [4], or Adobe Version Cue [1]. The distributed storage system described in this paper has three goals: to support file versioning, file sharing and endless files as defined later in this paper. An additional minor goal is to design the whole system without need for distributed locking while still avoiding race conditions.

We have two basic motivations for the *file versioning*. First, it is appropriate for lock-free design of the whole distributed system, because it avoids reader—writer conflicts. The second motivation is the fact that we are able to track history of changes to files and we are able to revert any changes. History tracking support is not limited only to text files, such as source codes, but it also applies to binary files such as images or even application-level metadata for other files.

The *file sharing* is usually natural requirement on a distributed file system. While read-only sharing does not pose any problem, read-write file sharing contradicts with file versioning by its common definitions. In this paper, we present a relaxed definition of read-write file sharing, that allows designing and implementing a system where both properties are available. Clearly, two distinct file systems may be developed, one with file sharing and the other one with file versioning or the file versioning may be offloaded to application level. Traditional file systems use static storage space allocation and therefore the storage capacity has to be split in advance between the shared and versioned file systems.

The application-level file versioning does not allow the data blocks to be shared among the file versions. We overcome these issues by using block-based storage system which enables us to allocate dynamic storage capacity. We show that the proposed distributed file system is capable of combining these two features inside a single directory structure and is capable of sharing the data blocks among the file versions.

Significant motivation for the *endless files support* can be found in applications that produce large continuous data flows, such as security or industry cameras recording and archiving. A common attribute of these use patterns is the natural expiration of data in time—very few people are actually interested in plain archive of such records and without expiration, such an archive would also run out of storage space eventually. Thus a better approach is archiving the interesting parts only, i.e. parts that someone has recognized, explicitly marked, and for example put into separate archive; while the actual data is kept for some limited amount of time only, for instance from now to a few hours in the past. We still want to support both file versioning and endless files within one file system but with a restriction that a single file may be either versioned or endless; which means that versioned endless files are not supported. The endless files are modified continuously and therefore it makes no sense to provide the file versioning for these files.

As mentioned above, another goal is to develop a distributed framework without requiring a distributed locking scheme. Utilization of append-only data blocks enables us to build completely lock-free system as there is no need for any distributed locking at storage level if data blocks may be written only once. The lock-free storage substrate also allows introducing disconnected operations that implicitly avoid starvation of locks and race conditions. An additional reason for append-only data blocks is that write once—read many (WORM) media type such as CD-R, DVD-R may be used for data blocks storage. This media type is suitable for storing the data which must be guaranteed to be immutable like auditing logs.

Another advantage of *generic block storage* substrate is its support for dynamic changes in storage size. New storage servers may be connected to increase capacity. Existing servers may be disconnected to shrink the total capacity, after the allocated data blocks are moved to the remaining servers if data loss is not acceptable and the data is not available in redundant copies.

The rest of this paper is organized as follows. In Section 2, we define the file versioning, the file sharing, and the endless files support more precisely and analyze difficulties in combining such features together. In Section 3, we present the proposed storage architecture from theoretical point of view and Section 4 discusses practical implementation, its properties and shortcomings. Section 5 briefs related storage and versioning systems, Section 6 discusses future directions of development of the presented system and Section 7 gives work summary and concluding remarks.

2 Principles of File Versioning and Data Sharing

2.1 File Versioning

We define file versioning as tracking history of changes to files, where each file has a version, initially set to a default value and increased after changing the file. After each version change, the file becomes immutable and changes are stored to a new file of the same name but with increased version number. Users access the last version of a file by default but they have tools to track history of changes and to access previous versions of a file. Once the file is opened, user accesses the particular version no matter how many new versions are created while the file is open. The history of versions can be unlimited, so that the system keeps all versions from an initial version to the current file, or it keeps only a limited number of versions. Usually the system keeps n latest versions of a particular file because of potentially huge history size unless WORM storage substrate is used.

2.2 File Versioning with Data Sharing

We define file sharing as an ability to read and/or write from/to the same file simultaneously. Read-only file sharing means that all users read particular file simultaneously and none of them modifies the file. Read-write sharing means that users can both read from and write to the file.

The novel approach presented in this paper is file sharing together with file versioning based on immutable files, or on lower level, based on immutable block storage. Read-only file sharing does not represent any problem, but immutable files are contradictory with common read-write file sharing. Immutable files require that once a file is written, it will not be changed any more, while read-write shared files require that a file can be changed and read arbitrarily. We combine file versioning with immutable files so that immutable files can be modified by creating a new version. We also allow for switching between read-write and versioned modes on an individual file, thus providing user with capability of creating “virtual snapshots” of the shared file.

Read-write sharing can present various race condition conflicts. We decided to offload solving of such conflicts into application level and it is up to clients to negotiate non-conflicting sharing access patterns.

2.3 Endless Files

We define endless file as a file which is continuously appended, e.g. live stream from camera. However, we have to limit such a file to avoid exceeding capacity of storage servers. We support endless files via *a sliding window* mechanism, which means that the beginning of such a file expires after some amount of time. More precisely, endless file is a file, which can be read from some offset to the actual file length and may not be versioned. The endless file is usually written in append-only mode, but we allow accessing the file in full read-write mode including shared read-write mode.

3 Storage Architecture

Our distributed storage system consists of two parts. The first part is distributed immutable storage substrate. The second part is a metadata manager that maps files to storage substrate and organizes files into directories. The data storage process is client driven, so that the client stores the data blocks to storage servers and then creates metadata file which is sent to the metadata manager. In the case of data retrieval, the client retrieves metadata file from metadata manager and then directly communicates with the storage servers. This does not guarantee that the client always accesses the very last version of a particular file. However, primary goal of versioned files is not to handle such race conditions. If this poses a problem for the client, shared files must be used instead or application-level locking needs to by applied.

3.1 Storage Substrate

Our distributed storage system uses block-oriented immutable storage provided by a group of servers that are known to clients through some service. Servers can be connected and disconnected at any time. Providing immutable block oriented storage means that the client can store only blocks of data that are immutable. If the client needs to change the data, a new data block must be allocated on a server. The client can request existing data blocks to be removed. Servers do reference counting on blocks, that they provide, based on information from clients and metadata manager. Thereafter, servers are able to decide whether particular data block can be physically removed or whether it is still used.

3.2 Metadata Manager

The metadata manager, that maps data blocks into files, is a second important part of our storage system. Metadata corresponds to I-node in UNIX-like file systems including the fact that the file name is not a part of metadata itself. In our system, metadata consists of identification of particular blocks which comprises location of data block on storage server (includes identification of the storage server and it is unique within the server) and a location of data block in a file (i.e. its offset and length). Unlike metadata in UNIX-like file system, we allow data blocks to be of arbitrary length, to overlap each other, and to be redundant. We define redundancy as allowing multiple data blocks, that contain the same data, to be mapped to the same offset of particular file.

The metadata manager maintains metadata, it maps metadata to file names and it also provides directory structure. In order to provide this functionality, the metadata manager offers the following basic operations that clients may use:

- **Store.** Clients store metadata on specified file path using this command.
- **Retrieve.** To retrieve metadata from specified file path.
- **List.** To retrieve names of directories and metadata files from specified path.

- **Remove.** Using this command, the clients request particular metadata file or directory to be removed. Metadata manager decreases reference count on all participating data blocks on storage servers. If the object being removed is a directory, then it must be empty.
- **Update.** For updating the content of a file, if file type supports updates.

3.3 File Versioning

As mentioned in the previous section, we assume immutable storage substrate, which naturally leads to data versioning. When clients want to make changes to their files consisting of data blocks, they must store the new data to the new data blocks. These new data blocks may be assembled in the new metadata together with old unmodified data blocks. At this point, the client has a new instance of metadata describing a new version of particular file. Reference counters of unmodified data blocks must be updated accordingly at the storage servers.

We need to define circumstances, under which a new version of a file is created. There are two basic approaches [8]: (i) a new version is created after each write to the file, or (ii) a new version is created after the file is closed. We use the latter approach, which is a reasonable compromise between granularity and size of the file history.

For two file versions u, v $(u \neq v)$, we define relation (u, v) which means that file version v is derived from version u and there does not exist $w_1, \ldots, w_n$ such that $(u, w_1), \ldots, (w_n, v)$, and there does not exist w and $\exists v_1, \ldots, v_n$ such that $(w, v_1), \ldots, (v_n, w)$, i.e. relations do not contain cycles. Let s be an initial file version, then for any file version v $\exists v_1, \ldots, v_n$ such that $(s, v_1), \ldots, (v_n, v)$. We can build *version tree* such that when u, v holds (u, v), u is direct predecessor of v in the tree. The root of such a tree corresponds to the initial file version. The depth of particular branch in tree represents number of changes to the file from the initial version to some particular version. We denote the set of *last file versions* as a set of leafs in a version tree. We can order the set of last file versions according to the file creation time. We denote the *latest version* as the version from the set of last file versions with the highest creation time.

We may want to limit the number of file versions. Therefore, we need to define which file versions are obsolete and can be removed. The most straightforward method, we can think of, is again to order file versions according to creation time and to keep only the n latest versions. Another possible method is to keep minimal subtree having at least n versions including the latest version. Using this method, we can mark a file obsolete, even if according to the time ordering the file is just behind the latest version. Using the former method, we can split version tree into forest (i.e. group of trees) and this can happen in the case that we mark obsolete a file version from which more then one new versions were derived. We decided to use the former method as it is easier to implement.

3.4 File Sharing

Within scope of this section, we use term *file sharing* as a shorthand for read-write file sharing. On one hand, we want a file modification to create a new file, but on the other hand, we want to be able to do parallel and multiple changes to a single file that is viewed by other clients. One of the possible ways to solve this dilemma is to divide files into shared and versioned, otherwise we lose file versioning semantics or file sharing semantics. Then a mechanism is needed to convert a versioned file into a shared file and vice versa. When we convert a versioned file into the shared file, we clone metadata of specified file version and we mark this file as shared. From this point on, we see the versioned file and the shared file as two independent files, which have the same content at the beginning. When we convert a shared file into the versioned file, we clone metadata of particular shared file and we create a new latest version from this shared file. From this point, the versioned file is immutable and it is further independent of its shared version, i.e. any change made into the shared file is not projected into the versioned file after the conversion. As we mentioned above, this approach can be used for creating snapshots of shared files and since we only clone metadata, which means that unmodified blocks are shared between versioned and shared file, this is a space-efficient approach.

3.5 Endless Files Support

We represent endless files using the sliding window approach. The window has a beginning o and its length is l. Client can read data anywhere in range $\langle o, o+l \rangle$. At a time T_i, we have a window $\langle o_i, o_i+l \rangle$. At a time T_{i+1}, client appends a bytes and we have a window $\langle o_i + a, o_i + a + l \rangle$. We need to define how we represent offset in this file from client's point of view.

If the client uses an absolute offset, we may run out of range of the variable representing the offset at some instant, as an endless file has potentially infinite offset and we are able to represent finite numbers only. Another problem with the absolute offset is that clients may get confused when accessing range $\langle 0, o \rangle$. Read access in this range does not have to be interpreted as an access fault, but in that case the client may make a wrong assumption that the file is corrupted for example. If we allow endless files to be supported not solely in append-only mode but also in writing mode in general, then write access in the range $\langle 0, o \rangle$ results in data loss as blocks expire immediately in this range.

If clients use offset relative to o (i.e. beginning of sliding window), there is a problem that the meaning of the offset changes with time. Consequently two clients are unable to synchronize themselves to a particular absolute offset.

We decided to use combination of both cases. The offset provided by the client is relative to defined fixed point of the client, which is implicitly zero and may be copied from the current value of o. This allows to avoid clients confusion from empty beginning of the file but it also allows clients to get synchronized through an absolute offset if necessary.

An endless file cannot be converted into a versioned file, at the data blocks of endless file expire, meaning that parts of corresponding versioned file would also expire and thus it would not be immutable. Clients may use classical copy command to create a versioned file from an endless file or its part but the classical copy does not allow sharing of data blocks among files.

4 Practical Implementation

4.1 Storage Substrate

Practical implementation of storage substrate is based upon network storage stack [2], which offers block oriented storage and is able to store byte arrays in the append-only mode. A part of network storage stack called the *ExNode* library collects allocated byte arrays to XML metadata file (corresponding to UNIX I-node). Using this ExNode library, clients can store and retrieve files to/from the IBP infrastructure. The IBP storage stack does not contain any metadata manager.

We have developed the *libxio* library, that provides standard UNIX I/O abstraction on top of the IBP storage stack. The initial version of this library [5] does not use any metadata manager and stores metadata files to client's local file system. The library uses special URI for accessing files instead of pure file name in the open(2) system call and this approach allows users to pass per-file options when accessing it. A part of the special URI is file access mode: versioned mode (by default), shared mode, or endless mode.

4.2 Metadata Manager

Metadata manager is a newly developed part of our distributed storage. The initial implementation does not contain distributed metadata manager but rather centralized for the sake of implementation simplicity. Together with libxio, it provides an application-level distributed file system; in the future we assume using a Linux kernel module for providing transparent file system.

Our initial implementation of the metadata manager does not implement full delete operation. If user requests a file to be deleted, only metadata is actually removed. Data blocks are not freed, which allows guaranteeing that data blocks are available to clients that opened the file before delete request and haven't closed it yet. Removing metadata means that the file cannot be re-opened which meets expected behavior. Such an approach requires an additional garbage collector. We use the garbage collector, which is a part of the IBP stack and frees the data blocks after certain amount of time called expiration period.

Metadata manager maintains directory structure and file names of files described by metadata in a local file system. For example, when the client wants to store the file */test/test1/test.txt*, metadata manager creates the directory */test/test1* on its local file system, where file with metadata named *test.txt* is stored.

File Versioning We said that metadata manager stores metadata according to its file path and file name. This approach needs to be enhanced to store versioned files. We have chosen the approach where every version of metadata is stored to a separate file on metadata manager. Therefore, metadata manager creates directory even for an individual file name and it stores individual file versions into this directory. This approach is demonstrated in Figure 1, where versioned files *test1* and *test2* stored in the directory *tmp* are shown. We store version number of the predecessor for every file, an initial version has *nil* predecessor. This enables maintenance of version trees.

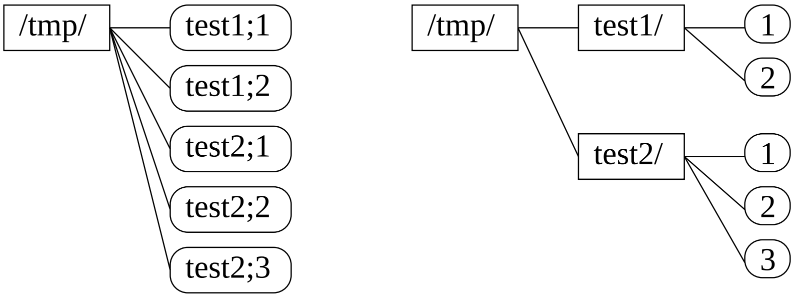

Fig. 1. Versioning representation. Sharp boxes represent directories, while round boxes represent individual files. Logical view is shown on the left side and physical view is on the right side.

We allow users to access any version of a file in both read and write modes. The *retrieve* command delivers the last version of the file if a version specification has been omitted and the specified version otherwise. The *store* command does not support version specification and always stores a file as a new version with the lowest unused version number. If the client accesses a particular file version in read-write mode, the reads are done from that particular file but writes are performed in a new version after the file is closed. Files in this mode can be accessed asynchronously, i.e. it is possible to use prefetch and a write-back cache.

File Versioning with Data Sharing We store versioned files by default. If we convert a versioned file into the shared file, we copy versioned metadata into the shared subdirectory. E.g. if we convert the file */tmp/test1/2* into a shared file then we copy this file into file */tmp/test1/shared/2*. This means that any file version can be shared only once and then it must be unshared before new sharing is possible. Unsharing copies the shared metadata into a new latest version of the file.

We need to provide more operations to the metadata manager to be able to handle shared files together with versioned files, namely operations for changing file mode (i.e. shared or versioned) and an operation for creating file with mode different than the default one (e.g. shared in our case).

For shared files, clients are required to fetch actual metadata before reading data blocks from storage servers, meaning that files are accessed synchronously. This may pose performance problems as the client must read and parse metadata file before reading data block. This can be sped up by introducing serial numbers in metadata, which are increased by every metadata change. In such a case, metadata manager can be extended with another operation for retrieving a metadata serial number, and the client just compares serial numbers instead of fetching and interpreting metadata. Another performance improvement is possible using big-enough size of a data block. A client is requested to store actual metadata to metadata server after it performs a set of write operations that produce a new data block.

Endless Files In the case of endless files, the metadata manager needs to provide an update operation that is atomic. The metadata manager stores metadata for particular file on single server, which allows the metadata manager to guarantee the atomicity of the operation without the need for global distributed locking.

As we said, we use offsets relative to some fixed point, by default set to zero. For offsets we use variable type of size 64 bits, which seems to be enough as 64 bits may overflow after about 200 years if continuous storing is done at the rate of 20 Gbps.

Another problem is representation of metadata. We keep information about offsets and length of data blocks and we have two possibilities here. We can store absolute offset of data blocks, which is limited by variable type used, or we may store relative offset of blocks (relative to offset o). The second case requires updating offsets of all data blocks in metadata, after the data block at the beginning has expired, and thus we have decided to use absolute offsets.

5 Preliminary Tests

Our preliminary testbed consists of two network setups: performance tests over the short distance (local area) network and over the long distance (wide area) network. In both cases, we use servers based on Pentium 4@2.4 GHz processor, 1 GB RAM, 1 Gbps NIC (Intel). The server used for metadata manager is equipped with 8 SCSI disks (73 GB each) assembled in hardware RAID 5 array (Adaptec 2200S ASR card). The servers used for IBP data storage are equipped with 8 to 10 SATA disks (250 GB each) assembled in hardware RAID 5 array (3ware Escalade cards). All clients are using a stock ATA disk.

In the short distance (local area) network setup, we were able to achieve 930 Mbps data transfer rate over a single TCP connection and 950 Mbps data transfer rate over an UDP connection, both measured with the *iperf* [6] tool. The corresponding data rates for long distance (wide area—in our case distance of 300 km) network were 220 Mbps for a single TCP stream with 2 MB TCP window and 930 Mbps for a single UDP stream.

We use the modified *tar* program to extract files into IBP depots using our metadata manager. We probe our system behavior using four types of files:

- File of 500 MB size to test medium size files.
- File of 10 kB size to test small size files.
- File of 0 kB size to test metadata manager itself.

– Linux kernel sources to duplicate real usage pattern including directories.

The Linux kernel sources contain 19000 files in 1200 directories of total size 218 MB. Since a file of size 10 kB and 0 kB is transferred too fast, we probed 1000 files of size 10 kB stored into a single tar archive and 10000 files of size 0 kB also stored into a single tar archive.

We choose the NFS file system as the reference although it uses UDP datagrams instead of TCP stream by default (IBP is TCP based). The IBP can use many storage servers while NFS v3 is unable to use multiple storage servers.

We can see performance of the transfers in the table 1. In the case of medium sized file, NFS and IBP are quite comparable, although using our experimental IBP stack [1], we are able to achieve time around 9 seconds for 500 MB file in both the local and the wide area setup. We have not implemented this experimental IBP stack into `libxio` yet but we believe there is a great performance potential. In the case of 10 kB files, NFS seems to be quite better than our system using IBP over a local area connection while IBP is slightly better over a wide area connection. We believe that a better IBP stack could help to achieve better performance. Also our system does not provide any local write back cache, each operation is synchronous. The 0 kB files show performance of our metadata manager compared to NFS's ability to create empty files. NFS seems to be slightly better than our metadata manager over a local area connection while our metadata manager is slightly better over a wide area connection. However, our initial implementation of the metadata manager is not highly speed-optimized. In the case of the Linux kernel sources, we can see that our system is comparable to NFS in both cases but the result for multiple storage systems clearly demonstrates the potential of parallel processing.

File type		**Local connection**	**Wide connection**	**8 IBP servers**
500 MB	NFS	21.4sec	23.7sec	
		186 Mbps	168 Mbps	
	IBP	19sec	18.9sec	16sec
		210 Mbps	211 Mbps	250 Mbps
10 kB	NFS	2.2sec	52.9 sec	
		36 Mbps	1.5 Mbps	
	IBP	45sec	44.1 sec	42sec
		1.7 Mbps	1.8 Mbps	1.9 Mbps
0 kB	NFS	18.8sec	4min 51.9sec	
	IBP	29sec	2min 37.3sec	
Linux kernel	NFS	6min 28sec	23min 3.5sec	
		4.5 Mbps	1.2 Mbps	
	IBP	8min 59sec	23min 29sec	8min 59sec
		3.2 Mbps	1.2 Mbps	3.2 Mbps

Table 1. File transfer time

[1] Our experimental and simplified reimplementation of the original IBP stack

In the table 2, we can see amount of disk storage needed for metadata and data. The metadata manager stores metadata into a directory structure. We archived this structure using *tar* and then count the size of each uncompressed archive.

	500 MB file	1000 10 kB files	10000 0 kB files	Linux kernel sources
Metadata size	60 kB	3 MB	20 MB	56 MB
Data size	500 MB	10 MB	0 MB	218 MB

Table 2. Data and Metadata size

6 Related Work

There are many general-purpose networked or distributed file systems such as NFS [11], DFS [9], AFS [10], and others, but there are only few systems that hit our target features. The Eliot [12] file system is built on top of immutable peer-to-peer storage. It uses immutable peer-to-peer substrate together with a metadata service that handles the mutable parts of the file system (i.e. inodes, symlinks, directories). The Eliot file system provides either a NFS or AFS-like interface and it tends to be a generic file system supporting neither the file versioning nor the endless files.

Versioned file system was introduced in OpenVMS file system Files-11 [13]. This file system is neither distributed nor networked and it uses simple versioning scheme—after each write operation, a new file version is created. The number of versions is limited. An upcoming database based file system from Microsoft—WinFS [14] should support file versioning. This file system is proprietary and had not been finished yet, therefore no more details are known to the author. Our system supports not only file versioning, but also immutable substrate and endless files.

Application-level versioned file systems are widely used mainly by software developer community, based on tools like CVS [3], SubVersion [4], or Microsoft Visual SourceSafe [7]. CVS system is used mainly for text files, it supports versioning trees, it can automatically merge conflicting updates. Subversion is another versioning tool similar to the CVS, inheriting most of the features from CVS and adding a few others such as versioning of directories, file renaming and metadata versioning. MS SourceSafe is again a similar tool on commercial basis. Until recently, the versioning was not widespread beyond software developer and scientific community—but now tools like Adobe VersionCue [1] working with images or binary files in general are available to completely different communities such as graphics designers.

File systems and tools supporting file versioning do not support file sharing. There is no file system known to the author that supports both file sharing and file versioning. Also there is no file system known to the author has even the endless files support.

7 Future Work

Our further work will be oriented primarily on enhancing the metadata manager. We are focusing on developing a fully distributed version of the metadata manager using a distribution function, which stores a particular file to a particular metadata server, possibly with redundancy support. Further enhancement will be in using a dynamic distribution function that allows us to connect and disconnect servers with running metadata managers. As discussed above, the metadata manager does not support operations such as full delete or rename and we want to add support for these operations while still preserving lock-free characteristics of whole system.

As mentioned in the Introduction, the lock-free designing allows for straightforward accommodation of disconnected operations and we aim to add disconnected operations support in future versions. For easier access to the data on our storage system, support for this distributed framework needs to be implemented into the operating system kernel, so that users could access data through ordinary file system interface transparently.

We know that any resilient distributed system that allows connecting and disconnecting storage servers needs to support redundancy, meaning that data from a single files is replicated across multiple storage servers. Our further research will be the support for voluntary data redundancy for both robustness and performance reasons.

8 Conclusions

In this paper, we are demonstrating the concept of distributed file system that supports file versioning, read-write file sharing, and endless files based on sliding window approach. It is based on distributed storage substrate providing immutable block allocations, and metadata manager, that maps files to data blocks using metadata and maintains directory structure. While file versioning and read-write sharing is often understood as contradictory, we demonstrate the approach that allows both properties to coexist. We demonstrate the idea of the endless files that could have a great potential for scientific and multimedia applications. We have described the application-level prototype implementation, that we use for verifying this concept. Prototype implementation is based on network storage stack and libxio library. While the prototype is already usable now, we envision a great potential for future development of the whole system.

Acknowledgments

This research is supported by a research intent “Optical Network of National Research and Its New Applications” (MŠM 6383917201) and by CESNET Development Fund project 172/2005. We would also like to thank to Luděk Matyska and Petr Holub for kindly supporting our work and for stimulating discussions.

References

1. Adobe Version Cue.
http://www.adobe.com/products/creativesuite/versioncue.html.
2. M. Beck, T. Moore, and J. S. Planck. An end-to-end approach to globally scalable network storage. In *SIGCOMM'02*, 2002.
3. B. Berliner and J. Polk. Concurrent Versions System (CVS), 2001.
http://www.cvshome.org.
4. Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version Control with Subversion, 2004.
http://svnbook.red-bean.com/en/1.0/svn-book.html.
5. Lukáš Hejtmánek and Petr Holub. IBP deployment tests and integration with DiDaS project. Technical report, Cesnet, December 2003.
http://www.cesnet.cz/doc/techzpravy/2003/ibpdidas/.
6. Iperf. http://dast.nlanr.net/Projects/Iperf.
7. Microsoft Visual SourceSafe.
http://msdn.microsoft.com/vstudio/previous/ssafe.
8. Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A Versatile and User–Oriented Versioning File System. In *The Third USENIX Conference on File and Storage Technologies*, 2004.
http://www.fsl.cs.sunysb.edu/docs/versionfs-fast04.
9. Open Group DCE. http://www.opengroup.org/dce/.
10. Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. In *IEEE Computer*, volume 23, pages 9–21, May 1990.
http://www-2.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/docdir/scalable90.pdf.
11. S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol, April 2003.
http://www.zvon.org/tmRFC/RFC3530/Output/index.html.
12. C. Stein, M. Tucker, and M. Seltzer. Building a Reliable Mutable File System on Peer-To-Peer Storage. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 2002. http://citeseer.ist.psu.edu/557574.html.
13. Files-11, OpenVMS. http://en.wikipedia.org/wiki/OpenVMS_filesystem.
14. WinFS. http://msdn.microsoft.com/data/winfs/.