SECURE AND FAULT TOLERANT DISTRIBUTED FRAMEWORK WITH MOBILITY SUPPORT

Lukáš Hejtmánek
Institute of Computer Science,
Masaryk University,
Botanická 68a, 602 00 Brno,
Czech Republic
xhejtman@mail.muni.cz

Abstract

In this paper, we propose an architecture of distributed data storage framework that incorporates fault tolerance, mobility support, and security. Main goal of our system is to provide equal opportunities for both connected and disconnected clients. Consequence is that mutual exclusion may not be involved. Data storage systems without mutual exclusion suffer from update and name conflicts. We avoid the update conflicts using immutable data storage. Mutable data is provided via either file versioning or Redo Logs. The name conflicts are automatically resolved without user's guidance, the file names are automatically changed to non-conflicting names, the directories are represented implicitly and thus we avoid conflict names connected to directories. Because the file names may be changed by the system, each file version is assigned an immutable globally unique identifier using which the file version can be accessed. Security model is based on certificates and VOMS attributes. This system is suitable for use within Grid VOs and it also supports services provided simultaneously to different VOs. Our prototype implementation exhibits very favorable performance so that it could be used as robust, secure, highly reliable and high performance Storage Elements.

Keywords: Distributed data storage, Logistical Networking, Security, Fault tolerance, Disconnected operations

1. Introduction

Large scale distributed systems consisting of thousands of nodes have serious problems with failures. If such systems have a large number of components (disks in disk arrays) and their total capacity is in the order of petabytes, then according to [27], they face component failure once a day. Fault tolerance can be achieved using some kind of redundancy. The redundancy can be achieved in a number of ways, starting with replication that is space but not CPU consuming, and ending with space conserving error correction codes [14]which require a lot of CPU power and furthermore, data modification is a quite complex and expensive operation. However, error correction codes are not commonly used in large scale distributed systems because firstly they are expensive to update and secondly, failure is the rule rather than an exception. In our approach, we opted for replication, with data stored in multiple identical copies. It is CPU conservative and it does not require extensive updating operations.

On the other hand, even simple a replication brings complexity into the architecture of distributed systems. Replication of read only data is trivial, whereas replicating mutable data brings problems with the consistency of the replicas. There are several approaches to dealing with replication and this consistency problem. We can divide replication into a primary backup approach and a state machines approach [24]. Primary backup approach refers to storing data on a fixed replica and this fixed replica distributes data to the other replicas. In the state machines approach data is stored directly to all replicas. It also means that replication is client driven whereas primary backups can be both client driven or server driven. The primary backup approach has a fundamental problem with a single point of failure. If a primary copy (i.e., the node that spreads data to the other replicas) is not accessible (e.g., due to network partitioning) then data updating is not possible. The state machines approach has a problem with data synchronization in network partitioning, as data updates in distinct network partitions may lead to different data states on particular nodes—we call this situation an *update conflict*. We can see that these two approaches represent a trade off between low availability (conflict avoidance) and low coherency (conflict resolution). A combination of the state machine and primary backup approaches is called the multiple-primary backup. In this case, we define a set of primary servers that are kept consistent using the sate machines approach and they collectively spread data to other replicas. This approach does not contain a single point of failure and users do not

need to upload data to all replicas. In this case, consistency is limited similarly to the state machines approach.

All the above presented approaches may be combined with two other concepts: pessimistic and optimistic replication [16]. Using pessimistic replication, data is spread synchronously to all replicas (and replicas are locked meantime, i.e., we mutually exclude concurrent updates). Using optimistic replication, data is spread asynchronously to all replicas and no exclusion is needed. Even these approaches represent yet more trade off between low availability (pessimistic replication) and low coherency (optimistic replication).

Data redundancy on storage servers is not a panacea for failures. When network failure occurs between the client and the distributed storage system, data redundancy on storage servers does not help. Therefore, we need to include the clients into replication system where the clients act as partial replicas of the storage servers. To completely conceal network failures, the client's replica has to provide all operations needed by the client, with the exception that only locally satisfiable requests can be completed. The same functionality can be also used to support a client's mobility. Mobility (or disconnected operations support) refers to the ability of a frequent connection and disconnection of the client and mainly the ability to work with data even when no connection is available. An example of such a system is CVS [3]where the users may check out files, disconnect from the network, work with files, connect to the network and commit changes. However, this mobility involves one fundamental problem connected to the fact that there is no upper limit on the duration of the disconnection. Without this limit, we cannot use any kind of mutual exclusion for conflict avoidance because prospective locks are either potentially held forever or prematurely released. Another challenge connected to mobility support is how to provide equal opportunities for connected and disconnected clients (with the obvious exception that disconnected clients cannot access arbitrary data but only data marked as accessible in disconnected mode), e.g., the disconnected clients may create new files and create new directories. These operations usually require mutual exclusion to avoid the creation of multiple directory entries of the same name. In this paper, we provide an approach to dealing with these problems and we provide a distributed framework with unrestricted mobility support.

Our aim is to build a large scale distributed storage system that provides (1) fault tolerance, i.e., it provides data replication, (2) mobility support, i.e., the system provides disconnected operations and mainly offers equal opportunities to both connected and disconnected clients, and (3) reasonable security model with properly authenticated and au-

thorized users. We expect that our work can be highly usable in Grid environments as secure, robust, highly reliable and high performance Storage Elements [6].

The rest of this paper is organized as follows. In the Section 2, we discuss the design principles of our proposed distributed data storage framework. Section 3 describes some preliminary experiments. In the Section 4, we discuss related work. Concluding remarks are given in Section 5.

2. Design Principles

Our goal is to provide a storage architecture where data is highly available and coherent. As we stated above, these two requirements are contradictory except in one case—immutable data storage, where data can be written only once and read many times (WORM, write once read many). For this reason, we have chosen a storage substrate that provides Write Once Read Many semantics of data storing. On this storage substrate, we aim to build a mutable data storage system that preserves data availability and data coherency. This concept of conflicts avoidance applies equally to mobility support as it does not involve mutual exclusion.

2.1 Immutable Data Block Substrate

In essence, our storage framework works with files. Files are decomposed into data blocks and metadata. The metadata contains references to data blocks and basically represents the files, it is equivalent to the well known UNIX I-node, the main difference being that data blocks are distributed across many storage nodes instead of being stored on a local disk.

Replicating read only data blocks does not pose any problem. We can adopt replication strategies mentioned in the introduction which would provide higher availability and low data consistency: data inconsistency is an issue of concurrent updates of data. In dealing with immutable data, data consistency is not an issue. We use the multiple-primary backup approach for data replication. This means that users may upload a data block to any storage node and may request the storage node to replicate the data block to another storage node.

Such an approach has two advantages. Firstly, the user does not need to upload data blocks to all replicas which could overload his network connection which can have lower bandwidth than network connection between storage servers. Secondly, there is no single point of failure as a client may contact any storage node. Because data blocks cannot

be updated, the data coherency is automatically provided. We enforce strict ordering of data and metadata, i.e., we add references in metadata to data blocks if the data blocks are completely stored on storage servers. Consequently, a referenced data block (which may be a replica of another data block) always exists on the storage server.

We need metadata replication for two reasons. The first is that the clients can cache metadata so that the clients do not overload metadata servers and the cached metadata is basically a replica of metadata. And the second is that files are not accessible without metadata, thus non replicated metadata forms a crucial single point of failure. However, replicating metadata is not without its problems. We have stated that metadata contains references to data blocks. If we replicate data blocks, we add more references to the corresponding metadata. This means that creation (or deletion) of data blocks changes metadata, and consequently, the metadata is no longer read only which means that it cannot be easily replicated. On the other hand, using simple consistency vectors[1], we can easily detect metadata changes and in particular all of the metadata can always be merged into a single file. This is due to the fact that metadata can be seen as a set of data blocks. We can merge metadata using a standard union operation to all the sets. The only problem here is to distinguish between the addition and removal of a data block, but this can be solved using the above mentioned consistency vectors.

Using the principles outlined above, we are able to provide distributed and replicated immutable data storage. While immutable storage is quite suitable for distribution and replication, it is quite unsuitable for users. Therefore, we show the way how to provide mutable distributed and replicated data storage on top of immutable substrate.

2.2 Mutable Data Storage with an Immutable Data Block Substrate

Providing a mutable file system on top of the immutable storage substrate involves creating a new file whenever a block is changed. The new data blocks must be added to the corresponding metadata which results in a change in the metadata. In the previous subsection, we stated that mutable metadata does not pose a problem, while the mutation of metadata is caused by addition or removal of replicated data blocks. This

[1]We use the term *consistency vectors* instead of the standard term *version vectors* [13]as we use the term version for file versioning. A mutable object is assigned a serial number. We increase the serial number with each object modification and we also maintain the previous serial numbers. Using serial numbers of the object and their history, we are easily able to detect changes in replicated objects.

property does not hold for mutations caused by file updates. Concurrent updates of the same file may cause update conflicts that may be difficult to resolve (user guidance may be required). Consequently, immutable metadata is needed to avoid update conflicts (immutable metadata with the exception of adding or removing data blocks replica). However, the immutable metadata imposes immutability also on files.

For immutable metadata, we can use the same approach as for the read only data blocks. Updating the file results in new metadata. Such approach basically creates a form of file versioning. Each set of metadata for a particular file corresponds to a particular file version. Taken together, we present the so called *versioned files* that are mutable files consisting of immutable file versions. File versions are represented by immutable metadata that references immutable data blocks. Every update of a versioned file results in a new file version, i.e., in new data blocks and in new metadata. We can use a replication strategy that provides high availability of data but must deal with update conflicts. Using such a strategy, we provide high availability of data which avoids update conflicts by read only data and we still provide mutable files. To avoid an explosion of version numbers of versioned files, we use open-close semantics. This means that the new file version is created only after the file is closed.

However, two problems still remain: (1) These principles do not deal with name conflicts—i.e., conflicts caused by the creation of multiple directory entries with the same name. We could use directory versioning which solves the problem but this is characterized by an explosion of number of directory versions. Changing any directory entry increases a number of directory versions up to the root directory. (2) Another problem is how to represent non-versioned mutable files. A new file version is usually created after the file is closed which makes it impossible to (read/write) share files between parallel applications. Both problems are addressed in the following sections.

2.2.1 Name Conflicts. There are many approaches to dealing with name conflicts but basically we can divide them into two groups: (1) conflict avoidance and (2) conflict resolution. Name conflicts can be avoided by using either a mutual exclusion or a read only approach. We have stated that versioning is not suitable for directories due to versions explosion. On the other hand, mutual exclusion is not suitable for mobility unless we impose an upper limit to the duration of the disconnected state. Consequently, we must use one of the conflict resolution strategies.

Name conflicts that arise from file name operation such as create and rename are solved using automatic renaming. Such a conflict is detected either when the new name is created or when the client switchs from the disconnected mode to the connected mode. During this transition, data and metadata are synchronized with storage servers. If the client created a new file name, it is created on the storage servers too and it may cause a name conflict. The conflicting name is automatically changed. For instance, three conflicting names `testfile.txt` are automatically resolved into two new names `testfile.txt#1` and `testfile.txt#2` (one of the files keeps the original conflicting name). The consequence of this approach is that the file name cannot be used as an immutable file identifier. Thus, beside file name, each file (and each file version) is assigned a globally unique identifier that can be used to access the file instead of using the file name.

Replication of versioned files leads to special name conflicts. We distinguish versions via numbering them and we use a deterministic algorithm to assign file version numbers[2]. This algorithm runs at each replica (storage server) and is local to that replica. It may happen (after concurrent updates of the same file version) that two instances of the algorithm, each running on different replica, assign the same version number to file versions with different content. We solve such name conflict by using our replica synchronization algorithm [8]. This algorithm does not involve mutual exclusion which would cause the non-availability of data or metadata, but it is still able to guarantee identical version numbers for the same file version on all replicas.

Similarly, name conflicts may arise from directory name operations such as create and rename. Also in this case, these conflicts are usually avoided using mutual exclusion. We avoid directory name conflicts via implicit directory representation. This means that we use a flat directory structure and a full path is an attribute of a file. The downside of this approach lies in the absence of authorization information bound to directories. Authorization information can be associated with files only and the user cannot create directories exclusively for himself. On the other hand, we believe that extended ACLs for files can mostly substitute ACLs for directories, this is discussed in more details in Section 2.4.

2.2.2 Non Versioned Mutable Files. Our system does not support mutable files in their natural way. We simulate mutable files

[2]We increment the last version by one to get the new last version. The increment is made locally on the replica thus it is possible that two or more replicas assign the same number to different file versions.

via versioned files. However, versioned files with open-close semantics cannot be shared between applications that update the file in parallel. We represent mutable files as versioned files but with changed semantics. Within this changed semantics, the new file version is created either (1) after predefined timeout, or (2) after predefined amount of new data, or (3) after the mutable file is closed. It is clear that using this extended semantics, the number of file versions rapidly grows. To avoid an explosion of version numbers, we remove obsolete versions, i.e., file versions which are completely overwritten by newer versions.

If storage servers are reachable, the client checks before each read or write operation whether a new file version is available. If it is available, the client downloads and uses the new metadata (file version). Using this approach together with the extended access semantics described above, updates are distributed among other online clients within a predefined timeout.

To avoid disk space wastage caused by a potentially large number of file versions and also due to the fact that file updates are usually small compared to the overall size of the file, we store initial file versions and then we store only updated records, i.e., the differences from the previous file version. This approach is the well-known log structured file system approach that use Redo Logs [15]which is a log of immutable update records. Each update record contains information about the update, the offset in file, and the length of the update.

However, using this concept of Redo Logs, we do not guarantee that the updates are instantly visible to all other participants. And because the files are mutable and we do not use mutual exclusion, update conflicts may arise after the concurrent modification of the same area of the same file. The conflicting updates are resolved automatically so that one of conflicting updates prevails the others are lost. We do not explicitly specify which update prevails and which one is lost. Neither do we guarantee that subsequent updates have the same order for all participants except two cases. The first case is, if all subsequent updates have been distributed in the same way through the network and the second is if a time period between two subsequent updates is higher than the time period required to distribute updates between all participants. If such behavior is not acceptable then the application level mutual exclusion should be used.

2.3 Metadata Handling

We store metadata on metadata servers. Metadata can be replicated and replication is done per versioned file (or its equivalent—non ver-

sioned mutable file). Replication of metadata is driven by a dynamically elected replication coordinator which is responsible for coordinating the replication for a single update of a single file. Further updates and different files can be coordinated by another coordinator. Replication runs asynchronously to updates. Our replication algorithm can be found in [8].

Metadata servers are distributed across a network. Metadata is spread among metadata servers using virtual distributed search tree P-Grid [1]. We have chosen this peer-to-peer system because it is possible for the clients to gain routing tables from the metadata servers and from which the clients are able to predict where the metadata is stored. This prediction is precise if the set of metadata is stable (without any metadata server connects or disconnects) which should be the case most of the time.

As stated above, we do not explicitly represent directories. Path names are an attribute of files but we use path and file name as a key to the P-Grid system to find file location. This approach is problematic for directory content listing because files of a single directory may be spread among many metadata servers. Thus, for directory listing, the client must contact all metadata servers. Solving this problem is one of our future tasks.

2.4 Security

We can say that file systems internally decompose files into data and metadata. In terms of UNIX-like file systems, we have I-nodes (metadata) and data blocks that are referenced by these I-nodes. Assuming that a user does not have direct access to raw storage media, the user cannot access data without knowing a particular I-node, therefore access control is usually made at this level. Once users are allowed to access the I-node, they are then granted access to the data. However, if we split metadata and data into two independent services, we must require access control verification at both services. We then face the problem of how to force a user to pass access control on both services in a defined order and how to verify that both services have granted or denied access.

The issues have been reduced to the following problems. We are given a set of services. We need to force a client to obtain a token from the services in a defined order given by service providers. We require that no service is skipped by the client and that the client cannot skip a service using an old token. Further, we require that the verification of a token is always local and no service is required to contact a third party during the verification process. The second problem is how to provide

cacheable time limited metadata to the clients. The metadata manager issues metadata to the client, so the client may cache metadata for an unlimited period, potentially, which makes authorization irrevocable. Since the size of the metadata is not insignificant, the creation of a signature can take significant period of time. Signed metadata must be valid only to the particular user and only for a specified time period. Signature must be certifiable offline. The solution to these problems is presented in [9]. This solution extends network storage stacks from logistical networking so that each part of the network storage stacks authenticates and authorizes the user. User authentication is based on PKI, authorization is based on ACLs.

As we do not explicitly represent directories, we cannot bind ACLs with them. ACLs bound to directories basically serve as shortcuts for setting appropriate ACLs to individual files. For instance, we may deny entry to a directory instead of denying access to the individual files. Thus, we can simulate directory ACLs by file ACLs except in two cases: (1) we cannot deny the creation of new files in a directory and (2) we cannot hide the subdirectories (which is usually done by denying directory listing). Neither we can deny entry into a directory but if we deny access to all files and set all files to be invisible (both possible using an appropriate file ACL) in a subtree beginning in this directory, the result is the same. Taken together, we believe that the inability to bind ACLs with directory does not impose a real problem.

3. Experiments

Our prototype implementation utilizes the IBP protocol from the Logistical Networking concept [2]. Using the IBP protocol, we build an immutable data blocks storage substrate. We are using our own implementation of the components of the Logistical Networking with an extended security model as described in [9]where also performance tests related to the extensions can be found. The IBP servers are implemented in C language. The IBP servers allow to store data blocks and allow modification of these data blocks but the latter feature is not utilized in our system. The metadata (called eXnodes in the Logistical Networking) is represented by XML files. The metadata is stored at metadata managers which are implemented in Java language. The client side of both IBP and metadata interfaces is implemented also in Java language.

Our experiments have been focused on file storage and file retrieval and their performance on high speed networks. We have used a single client equipped with 10 Gbps fibre optics network card, 8 GB RAM, and two dual-core Intel Pentium Xeon processors. We have used eight stor-

age servers, each equipped with 1 Gbps metallic network card, 8 GB RAM, and two dual-core Intel Pentium Xeon processors. The storage servers use disk array consisting of two 320 GB SAS disks organized as software RAID 0. We are able to store data into a single file at 139 MB/sec (1112 Mbps), and to read data from a single file at 178 MB/sec (1424 Mbps). Using `iperf` [10]network performance tool, we are able to achieve 750 Mbps between the client and any storage server using a single TCP stream. This limited transfer rate is caused by the network interface card at storage servers. However, using multiple TCP streams simultaneously from the client to all the storage servers, we can achieve aggregate rate of 5.6 Gbps. This special setup has been used to demonstrate that our system allows to utilize extensively the storage servers in parallel.

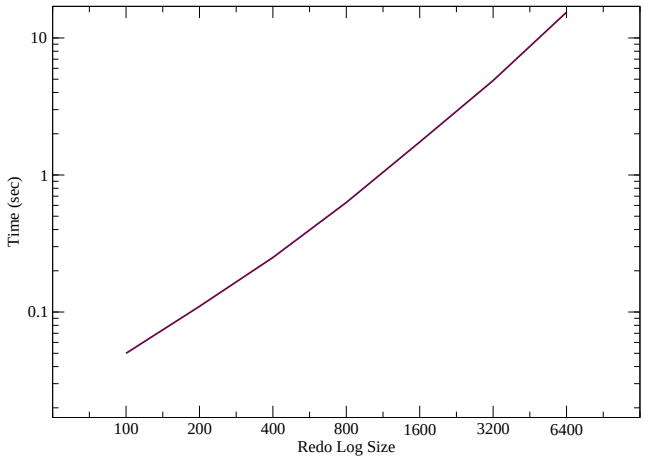
Table 1. A single file upload and download transfer rate and RAM and CPU usage. The usage and transfer rate is measured at the client.

Block Size	**File Size**	**Transfer Rate**	**RAM**	**CPU**
32 MB	156 GB	3656 Mb/sec down 3392 Mb/sec up	756 MB	90%
2 MB	9.7 GB	2488 Mb/sec down 2000 Mb/sec up	90 MB	50%

We did several tests to evaluate our prototype implementation. The first simple tests evaluated overall performance of storing and retrieving large files from and to client's memory only to eliminate client's local disk performance. We evaluated transfer speed for file upload and download, CPU usage, and RAM usage for two data block sizes: 2 MB and 32 MB. We expect that the latter size will likely be used. Each file comprised 10,000 data blocks. The results can be seen in Table 1. We can see that using 32 MB blocks, we are able to saturate available network bandwidth up to 65% (for download) and up to 60% (for upload). Using 2 MB blocks the bandwidth saturation is lower due to higher overhead when manipulating smaller data blocks. We can also see that larger data blocks require a lot of memory. To achieve such high transfer rates, the client must allocate at least two data blocks for each storage server, thus 512 MB is occupied by data blocks cache for the 32 MB data blocks and 32 MB for the 2 MB data blocks. Rest of the memory is occupied by the Java application itself. In the case of 2 MB data blocks, we can see lower CPU usage because the smaller blocks require more messages to be sent. This is because CPU is idle during the message sending process.

We stated that mutable files are represented by the Redo Log. It may happen that the Redo Log size is not negligible. The whole Redo Log is traversed and processed when the file is opened. Therefore, we have evaluated relation of the Redo Log size and the duration of file opening. The results can be found in Figure 1. We can see that up to 1,000 update records, the duration of file open is negligible. Assuming 32 MB blocks, 32 GB file is opened within 1 second. There is optimization possible as we could merge individual update records into a bigger single update record. This optimization is left as a future work.

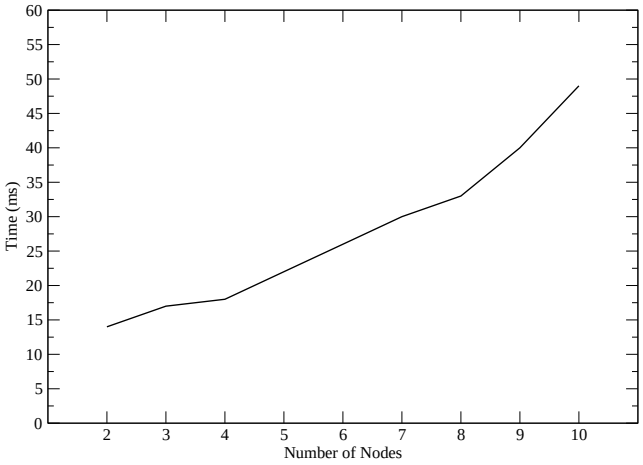
Figure 1. Duration of the file open operation for varying number of update records in the redo log.



We have not evaluated latency of replicated data blocks because the storage servers have relatively slow network connection to each other compared to the connection to the client. Thus data replication should be originated directly from the client and under such conditions, storing of two replicas of a file takes exactly once more time than a single replica of the file.

On the other hand, metadata replication is handled by our distributed algorithm and for this reason, we evaluated performance of distribution of the metadata updates. The results can be found in Figure 2. We can see that our algorithm scales well and that distribution of the metadata updates takes about 10% of time of data distribution. The distribution process runs asynchronously to update operation. The Figure illustrates time limit within which the metadata updates are distributed to all the replicas.

Figure 2. Latency of file update distribution among specified number of replicas.



Our prototype implementation has shown feasibility of our system. Our preliminary experiments have manifested that our system provides high performance distributed data storage. The clients can access the storage servers in parallel to utilize available network bandwidth. The experiments have also shown that some parts of the system will need optimization in future, such as opening of mutable files consisting of a large number of the update records.

4. Related Work

There exist many different distributed storage systems incorporating different approaches to the data storage problem. There are standard distributed file systems with POSIX access semantics such as AFS [18], NFSv4 [20], GPFS [19], Lustre [4]which, however, do not contain support for mobility and their support for replication is very limited (AFS—read only replication, NFSv4—incomplete specification, no complete implementation, GPFS—fixed number of replicas, Lustre—no replication of metadata servers).

The Coda [17]file system was one of the first file systems that presented disconnected operations. Compared to AFS, the Coda also provides read write optimistic replication. Replication granularity is per volume rather than per file. Volume is a set of files belonging to particular directory subtree. Coda also distinguishes between connected and disconnected mode, and it reports conflicting updates to the user. The security of the Coda file system is based on user IDs and group IDs.

In addition to these distributed file systems there exist experimental distributed storage systems which have either POSIX access semantics or access via special API.

Ceph [25]file system is replicated, scalable, and high performance distributed file system. It decomposes files into data blocks and metadata. Data blocks are stored to object storage devices (OSD). Data blocks are organized into placement groups and using CRUSH (Controlled Replication Under Scalable Hashing [26]) are mapped to OSDs. Compared to our system, this approach has the disadvantage, in that if we add more OSDs, some of the existing data blocks may be required to migrate to the new OSDs. Replication uses a primary copy approach using a monitor which coordinates the election of the primary copy holder. The monitor impose a single point of failure, on the other hand a monitor is not required if the primary copy holder is reachable. Ceph does not support mobility or file versioning. Security is based on time limited capabilities issued and signed by metadata servers.

Ivy [12]is a read write peer-to-peer file system. It uses DHash [5]peer-to-peer block storage substrate, and all data is stored as a value into a distributed hash using data checksum as its key. DHash provides replication of immutable data blocks. A mutable file system is provided via a log that forms a linked list of immutable log records. The user processes his own log and all publicly available logs and searches for the most recent changes. Compared to our system, Ivy provides only open-close access semantics. It does not explicitly support file versioning but it supports snapshots. It does not support mobility.

Eliot [21]is another peer-to-peer file system built on immutable peer-to-peer storage. It uses Charles [22]reliable and fault tolerant block storage substrate. But it uses only a single mutable metadata service which degrades Eliot fault tolerance.

The large scale storage system, OceanStore [11], provides file versioning (old versions are read only), disconnected operations and replication. However, performance is limited due to slow file lookup and also due to protocols for the Byzantine agreement. Compared to our system, the client is not allowed to predict data location and speed up metadata manipulation.

The Google file system [7]is an application level replicated distributed file system used for the well known Google search engine. Its architecture is based on a single master server (which imposes a single point of failure) and multiple chunk servers. The architecture is optimized for fast reading and appending files. Compared to our system, it does not provide file versioning or mobility support. Our system also supports replication of our equivalent to the Google's master server.

The L-Store [23]application level distributed file system closely resembles our distributed data storage system. It is also based on IBP [2]protocol. It supports replication of both data and metadata. Compared to our system, it does not provide file versioning or mobility support. Its security model requires online communication between a metadata server and an IBP server which makes it less robust compared to our security model.

5. Conclusions

We have designed a reliable, fault tolerant, and secure framework for distributed data storage with mobility support. The framework offers equal opportunities for both connected and disconnected clients which requires the system not to involve mutual exclusion. Systems without mutual exclusion suffer from update and name conflicts. We avoid the update conflicts using immutable data storage. Mutable data is provided via either file versioning or Redo Logs. The name conflicts are automatically resolved without manual guidance of a user, the file names are automatically changed to non-conflicting names, the directories are represented implicitly. Thus we avoid conflicts in directory naming. Security model is based on certificates and VOMS attributes. This makes the system suitable for use within Grid environments with the VO concept and it also supports services provided simultaneously to different VOs.

We have designed and implemented a prototype implementation and performed preliminary performance evaluation. This evaluation shows that even the prototype implementation exhibits very favorable performance so that it can be used as secure and high performance Storage Element service.

Acknowledgments

This research is supported by a research intent "Optical Network of National Research and Its New Applications" (MŠM 6383917201) and by the CESNET Development Fund project 172/2005. I would also like to thank to Luděk Matyska and to Petr Holub for stimulating discussions and help with the work described in this paper.

References

[1] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD*, 32(3):29–33, 2003.

[2] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. *SIGCOMM Comput. Commun. Rev.*, 32(4):339–346, 2002.

[3] Brian Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[4] Cluster File Systems, Inc. Selecting a Scalable Cluster File System, 2005. Cluster File Systems, Inc. Whitepaper.

[5] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM Press.

[6] EGEE. Site Access Control Architecture DJRA3.2. 2005. `https://edms.cern.ch/document/523948`.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[8] Lukáš Hejtmánek and Luděk Matyska. Distributed Data Storage with Strong Offline Access Support. In *The Second International Multi-Conference on Computing in the Global Information Technology Challanges for the Next Generation of IT & C*, pages 1–6. IEEE Computer Society Press, 2007.

[9] Lukáš Hejtmánek, Luděk Matyska, and Michal Procházka. Secure logistical networking in virtual organizations. Technical Report 2/2007, CESNET, z.s.p.o, February 2007.

[10] Iperf. `http://dast.nlanr.net/Projects/Iperf`.

[11] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *SIGPLAN Not.*, 35(11):190–201, November 2000.

[12] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer File System. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, 2002.

[13] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[14] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[15] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[16] Yasushi Saito and Marc Shapiro. Replication: Optimistic Approaches. Technical Report HPL-2002-33, HP Laboratories Palo Alto, 2002.

[17] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[18] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, 23(5):9–21, May 1990.

[19] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, Berkeley, CA, USA, 2002. USENIX Association.

[20] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC 3530: Network File System (NFS) version 4 Protocol, April 2003. `http://www.ietf.org/rfc/rfc3530.txt`.

[21] C. A. Stein, Michael J. Tucker, and Margo I. Seltzer. Building a Reliable Mutable File System on Peer-to-Peer Storage. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 324, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Lex Stein, Michael J. Tucker, and Margo I. Seltzer. Reliable and fault-tolerant peer-to-peer block storage. Technical Report HU-TR-04-02, Harvard CS, 2002.

[23] Alan Tackett, Bobby Brown, Laurence Dawson, Santiago de Ledesma, Dimple Kaul, Kelly McCaulley, and Suyra Pathak. QoS issues with the L-Store distributed file system, 2006. Advanced Computint Center for Research and Education, Vanderbilt University, Whitepaper.

[24] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[25] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320. USENIX, 2006.

[26] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Grid resource management—CRUSH: controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122, New York, NY, USA, 2006. ACM Press.

[27] Qin Xin, Ethan L. Miller, Thomas Schwarz, Darrell D. E. Long, Scott A. Brandt, and Witold Litwin. Reliability mechanisms for very large storage systems. In *MSS '03: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 146, Washington, DC, USA, 2003. IEEE Computer Society.