Churn Tolerant Virtual Organization File System for Grids *

Leif Lindbäck[1], Vladimir Vlassov[1], Shahab Mokarizadeh[1], and Gabriele Violino[2]**

[1] Royal Institute of Technology (KTH), Stockholm, Sweden
{leifl,vladv,shahabm}@kth.se
[2] Net Result AB, Stockholm, Sweden
gabriele.violino@gmail.com

Abstract. A Grid computing environment allows forming Virtual Organizations (VOs) to aggregate and share resources. We present a VO File System (VOFS) which is a VO-aware distributed file system that allows VO members to share files within a VO. VOFS supports access and location transparency by maintaining a common file namespace, which is decentralized to avoid a single point of failure in order to improve robustness of the file system. VOFS includes a P2P system of file servers, a VO membership service and a policy and role based security mechanism that protects the VO files from unauthorized access. VOFS can be mounted to a local file system in order to access files using a standard POSIX file API. VOFS can operate in a dynamic Grid environment (e.g. desktop Grids) since it is able to tolerate unplanned resource arrival and departure (churn) while maintaining a single uniform namespace. It supports transparent disconnected operations that allow the user to work on cached files while being disconnected. Furthermore, VOFS is a user level technique, and the current WebDAV-based VOFS prototype can operate under any operating system that has WebDAV mount support.

Key words: Grid file system; virtual organization; peer-to-peer; security; namespace

1 Introduction

A Grid computing environment allows forming Virtual Organizations (VOs). A VO is a virtualised collection of users or institutions that pools their resources into a single virtual administrative domain, for some common purpose. A VO File System (VOFS) aggregates data objects (files, directories and disk space) exposed by VO members. *Expose* here means a VOFS operation to assign a data object (a directory or a file on a VO member's computer) a logical name in the VOFS namespace and make it accessible via a VOFS server.

* This research is supported by the FP6 Project Grid4All funded by the European Commission (Contract IST-2006-034567).

** Gabriele Violino was at the Royal Institute of Technology while doing this work.

One major challenge in such a file system is namespace management. The namespace should allow uniform and globally unique path names to be associated with data objects wherever they are located in the Grid [1]. Uniform here means access and location transparency of exposed data objects, and the same view of the file system at all nodes. This requires mapping a *logical name* of a file in VOFS namespace to its physical location. The global nature of grids enforces logical names to be uniform across different administrative domains.

In this work we consider ad-hoc grids built of resources voluntarily donated by VO members. VOFS contains different types of data objects exposed by VO members to be shared within a VO. This paper proposes a user-level solution for implementation of VOFS that allows exposing data objects, transparent access to the objects, and maintains the uniform namespace in the presence of resource churn (node leaves, joins and failures). The proposed VOFS has the following features that make it useful in ad-hoc Grids to create and maintain work spaces by exposing and sharing data objects by different applications and VO members.

1. VOFS includes a security mechanism that protects exposed data objects from unauthorized access. It supports VO membership management, authentication and role-based authorization according to VO policies;
2. VOFS maintains a uniform namespace despite of the resource churn;
3. User level technique that allows ordinary applications (file clients) to access the VOFS using a standard POSIX file API, i.e. the applications do not need to be modified to access files exposed to VOFS;
4. VOFS is easy to use for non-experienced users;
5. VOFS can operate under any operating system that has WebDAV [2] mount support, e.g. MS Windows, Linux, Mac OS X. It has only been tested on Linux and MS Windows;
6. VOFS supports transparent disconnected operations that allow the user to work on cached files while being disconnected

2 Overview

This work builds on our previous work presented in [3] that proposed three ways of maintaining the namespace: a centralized name service; a distributed directory; and a DHT-based name service. In [3] we have presented VOFS with the centralized name service that has the major disadvantage to induce a single point of failure and a potential performance bottleneck. In this paper, we propose to build VOFS with a namespace maintained as a *distributed directory* where the namespace information is distributed among peers so that a peer knows location of *at least* those remote files which exposed under directories hosted by the peer. In this VOFS design every peer can potentially learn the entire namespace (i.e. location of exposed data objects) via a gossiping mechanism.

In the current VOFS design we consider only files and directories as data objects. The data objects can be exposed to any path in VOFS. An exposed directory offers disk space which is used by VO members to create new objects.

Exposing of a file or directory from the local node makes the data object accessible for VO members. Each peer runs a file server that provides and controls access to data objects exposed from the local node, see figure 1. Access to the exposed objects is achieved by mounting the local VOFS peer to a mount point, e.g. a local path. We use the WebDAV protocol [2] to access and transfer files between peers. Use of WebDAV allows accessing VOFS through any mount utility supporting WebDAV, e.g davfs2 [5] which offers a POSIX complaint API. Ones mounted, access to VOFS is no different from access to local file system.

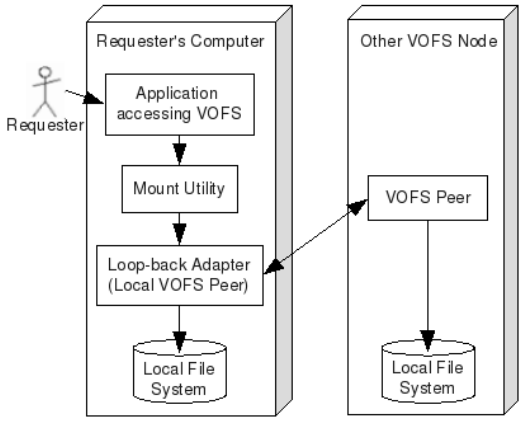

Fig. 1. Schematic view of VOFS architecture

3 VOFS Namespace and File Tree

VOFS is formed as an ordinary hierarchical file tree by exposing data objects in to the VOFS tree by given them VOFS names which are ordinary paths. The VOFS namespace is a set of mappings of logical names to physical locations. When a user exposes[3] a data object (a file or a directory) to the VOFS namespace hence the VOFS file tree, the exposed file is assigned a logical name, which is a path in VOFS. The path may include names of virtual directories. A *virtual directory* is not hosted by any peer, i.e. it does not really exist. Thus, VOFS consists of exposed real data objects (directories and files) and virtual directories that may contain other virtual directories and exposed real data objects.

Initially, the VOFS tree contains only the root, which is initially virtual. The VOFS namespace, hence the VOFS tree is formed explicitly and gradually as a result of exposing and un-exposing data objects.

Virtual directories help to maintain the VOFS namespace. If to assume that all directories in the VOFS tree are real (i.e. physically exist), then un-exposing

[3] The expose operation is described in Section 4.1

a real directory may cause partitioning of the tree as the data objects under the unexposed directory can not be properly identified. This also motivates introducing virtual directories. The unexposed real directory becomes virtual; and names of all objects under it remain unchanged.

It should be noted that when looking up location of a data object given its VOFS path, a longest prefix match is done taking the fully-specified path into consideration. The data object can be accessed if the exposing peer is online despite of whether other peers are online or not.

Mappings of logical names to physical locations are the major metadata of VOFS. The metadata associates exported data objects with a path in the VOFS namespace. It consists of logical name, physical address of a node and physical location of exported data object. The same meta-data are kept at every node in two tables: the `remote.db` table, which stores location information of data objects exposed by other peers; and the `local.db` table that stores location information of objects exposed by this peer. Whenever a data object is exposed, the exposing peer adds a pair of `local file system path` and `VOFS path` to the `local.db` table while all other peers will update their `remote.db` tables by adding a `VOFS path` and `physical host address` pair. Whenever a data object is unexposed, this information is removed from all peers. The namespace changes only when peers perform expose or unexposed operations. Peers communicate metadata by gossiping as explained below. All peers know the entire namespace, i.e which data objects are exposed and who exposes them.

3.1 Algorithm for Namespace Updates

To transfer namespace updates between peers we use a gossip algorithm based on the *lazy probabilistic broadcast algorithm* described in [7]. When a peer updates the namespace it sends an *update* message to all or some its neighbours. Each peer that receives an *update* message forwards it to all or some of its neighbours. There will be no loops since a peer never sends the same message twice.

There are no acknowledgements; instead the following recovery mechanism is used when messages are lost. Original sender id and a sequence number are attached to each message. Since there is FIFO delivery of messages, if a peer receives a message with a sequence number larger than the previous number plus one, it knows that some messages were lost. It will then send a *require* message to a subset of its neighbours. The *require* message indicates which message was lost and which peer is requiring it. A peer, which receives the *require* message checks if it has the required message. If yes, it sends the required *update* message to the requiring peer. If not, it forwards the *require* message to a subset of its neighbours. *Require* messages are forwarded only a specified number of times. Each peer maintains information about transmitted messages on its hard disk.

Note that the gossip algorithm described above is used only for namespace updates. All other communication, e.g. file transfer, involve only two peers.

Due to gossiping, there is no need to search for data objects since each peer maintains its own view of the namespace. The namespace view is almost the

same as views of other peers even though there might be some inconsistencies between views caused by update latency.

4 VOFS Peers

Each user who exposes data objects must run a VOFS peer on the users computer; while a user accessing VOFS does not need to run a VOFS peer. However, in the latter case, the user must know an address of any VOFS peer to be able to mount it and to access the VOFS. If the user runs a VOFS peer, then that local peer, *loopback adapter*, can be mounted to become the entry point to VOFS. In this case, there is no need to keep addresses of well-known mount points like in for example AFS [6]. Every of the VOFS peers provides the same set of the services that includes (un)expose, join, mount, cache. The services can be accessed by the user through the GUI of the VOFS peer. The services are described below.

4.1 (Un)Expose

A user (un)exposes data objects using an (un)expose client provided with a GUI in the current VOFS prototype. When exposing, the user defines a data object to be exposed and specifies its VOFS path. The expose services stores the logical-to-physical name mapping in the local table and initiates the update gossip algorithm. If the specified path does not exist, virtual directories are introduced in order to allow traversing the tree from root to the exposed data object. The root of VOFS is always `/`. It always exists at least virtually, but may also be mapped to a real directory. Name collision occurs when the user tries to assign a VOFS name which is already taken. In the current VOFS, the name collision is resolved as follows: if the data object to be exposed is a file, its mapping overrides the mapping of the object previously exposed with the same name; in case of directories exposed with the same name, their contents are merged.

4.2 Join

When a user starts a VOFS peer, the peer joins the P2P VOFS system. At startup, the peer downloads a list of all VO peers from the VO Membership Service (VOMS)[4]. Then the peer connects to some other peers selected from the list. The chosen peers and the new peer become neighbours. In the current VOFS prototype, selection of neighbours is random, but it could be done in a sophisticated way. They also exchange their VOFS views stored in their local and remote metadata tables described earlier. It is possible for the user to manually edit a peer's neighbour list through the GUI of the VOFS peer.

[4] described in Section 5.2

4.3 Mount

The user can mount VOFS with any mount utility supporting WebDAV used in the current VOFS; therefore we have not developed any special mount utility; instead, we use davfs [5] on Linux and NetDrive [8] on MS Windows. VOFS has not been tested on other OSs but Mac OS X has WebDAV support built in.

Once the VOFS is mounted, all POSIX file API is supported for manipulating data objects (provided the mount utility offers a POSIX API). The mount utility will translate the POSIX calls to WebDAV calls to the loopback adapter.

4.4 Cache

Each VOFS peer maintains a file cache. Read and write latency over network is compensated by the caching mechanism, which also allows offline work. VOFS uses *last write wins* reconciliation policy (a traditional file system policy for concurrent writes), which, if needed, can be replaced by a more sophisticated reconciliation policy implemented using, for example, Telex [9]. The cached copy is checked for update (compared to the master copy) when the file is read. When a file is written the new content is both stored in the cache and sent to the exposing peer, which informs all other peers who cached the file about the update. Also directory listings are cached, but unlike files they have an expiry time.

5 Security

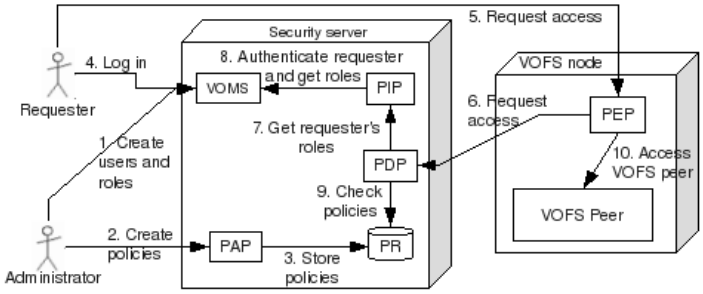

Fig. 2. Security components

The security infrastructure is based on the XACML authorization model [10]. Its goal is to provide authentication and authorization. When authenticating the users credentials are checked and the user gets a token which can be used to prove her identity in authorization checks. Authorization grants that users can only access resources to which they have right according to VO policies. Authorization is policy-based, policies are expressed in XACML.

5.1 Scenarios

A typical scenario of interactions between security components and VOFS peers is depicted in Figure 2. We distinguish four different phases: creating users and roles, creating security policies, authentication and access control.

Creating users and roles (1) The VO administrator uses the VO Membership Service, VOMS to create users and roles.

Setting policies (2) The administrator uses the Policy Administration Point, PAP to create policies. (3) The PAP stores the policies in the Policy Repository, PR. The PAP will invalidate the Policy Decision Point, PDP's cache. It can be specified in a policy when it is valid. This can be specified as time, date and day of week ranges and any combination of these.

Authentication (4) The requester logs in to the VOMS, using a web based interface. If the requester is authenticated, VOMS returns a token that is stored on the requesters's computer.

VOFS access (5) The requester uses an application that accesses VOFS. The mount utility sends the token along with the call to VOFS. The call is intercepted by the PEP which protects the VOFS peer. (6) The PEP asks the PDP whether the requester is allowed to access the peer. (7) The PDP asks the Policy Information Point, PIP for the requesters's roles. (8) The PIP contacts the VOMS to check if the token is valid and to get the user's roles. (9) The PDP evaluates the policies stored in PR. (10) If access was granted, the call is let through to the VOFS peer.

5.2 Components

The VOFS security infrastructure is built of the following components.

Virtual Organization Membership Service, VOMS keeps a database of users and roles in the VO. It has a web based management interface for updating this data. This interface is protected by a PEP. The VOMS is also responsible for authenticating users.

Policy Enforcement Point, PEP protects a resource (VOFS peer, VOMS, PAP). Each resource has a local PEP. The PEP sends authorization requests to the PDP and caches the answers. To improve performance the PDP answers not only to the request sent by the PEP, but to requests with the same subject and resource with all existing actions.

Policy Decision Point, PDP evaluates requests from PEPs according to the policies in PR. Policies are cached in memory. Invalidation of the PDP's cache also invalidates all PEP's caches.

Policy Information Point, PIP contacts VOMS to validate the requester's identifying token and get the requester's roles. The answer from VOMS is cached, together with the lifetime of the token.

Policy Administration Point, PAP is a server that makes updates to PR. The PAP is protected by a PEP. A client to the PAP is provided.

Policy Repository, PR stores the policies as XACML files.

We suppose that except for PEP there will be only one instance of each component per VO. Each PEP should be placed on the same host as the resource the PEP protects.

5.3 Secure Communication

The goals of secure communication are

1. To guarantee that PEPs get answers from the correct PDP;
2. To guarantee that PDP gets answer from the correct VOMS;
3. To guarantee that the token identifying a user is not stolen. If it is stolen it can be used to impersonate that user.

The first two goals can be met using certificates to identify PDP and VOMS. Regarding the third goal, there are the following risks that the token is stolen:

1. During transfer (this risk is eliminated with encrypted communication);
2. From the user's computer;
3. By a malicious node pretending to be a VOFS peer;
4. By another VOFS peer.

The second risk can be reduced if the VOMS encrypts the token with the user's public key. Before the token is passed to another peer it is decrypted with the user's private key. This means it is not possible to steal the token from a file in the user's local file system, unless also the user's private key is stolen.

The third risk is that someone writes a program that is not a VOFS peer but can issue correct commands to join the VOFS. If other peers believe it is part of the VOFS and communicates with it, it will get tokens of other users. This risk is eliminated if peers only communicate with other peers that can prove they are allowed by VOMS to take part in VOFS. To achieve this it is necessary that all peers can prove their identity using a certificate signed by a trusted certificate authority, CA. Such a certificate will contain the host address of the peer and will be issued by the VOMS that runs the trusted CA. Each peer will get its certificate from the VOMS at startup, VOMS maintains a list of allowed peers.

The fourth risk is that a trusted VOFS peer is compromised by a malicious user that change it to report calling peer's tokens. This risk can not be eliminated since the purpose of passing the token *is* to let the receiving peer impersonate the user of the calling peer. The risk can be reduced in the same way as it is reduced using proxy certificates [11], by restricting the life time of the token and by delegating only a subset of the delegator's rights.

None of the above solutions require the user to be aware that certificates are used. This makes the VOFS easy to use also for non-experienced users.

6 Implementation of VOFS prototype

The prototype can be executed on all platforms supporting Java Servlets. Figure 3 shows its main components. The prototype is bundled with Apache Tomcat. All that is needed to start it is to specify the PDP location and to start tomcat.

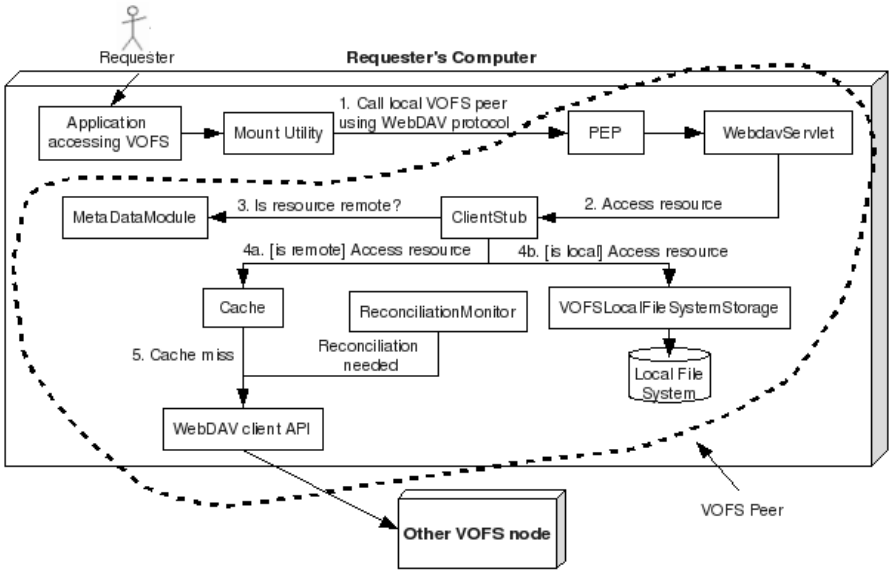

Fig. 3. VOFS implementation

PEP is a servlet filter that intercepts all incoming requests. It translates the WebDAV method of the call to a VOFS operation and calls PDP to check if the operation is allowed. If not, an HTTP 403 (forbidden) code is returned.

WebdavServlet The access point for remote peers and the local mount utility.

ClientStub Receives requests from WebdavServlet and forwards them to the correct component.

MetaDataModule Keeps meta-data, see section 3, and offers the longest prefix matching engine.

LocalFileSystemStorage Provides access to exposed files and directories.

Cache Caches remote data objects. If a searched object is not in the cache the call is forwarded to the remote peer hosting it. The returned object is cached.

WebDAV client api Responsible for contacting remote peers to read or write data objects.

ReconciliationMonitor Continuously monitors cache to see if an item in cache is newer than the master, if so updates the master.

7 Related Work

Sprite Network File System [12] is a distributed file system similar in some aspects to VOFS. Meta-data (location information) in VOFS with decentralized name service is handled in a similar way to Sprite. However; the scopes of the two file systems are different: Sprite is designed to operate within LANs; whereas VOFS should operate over WANs. A main difference between VOFS and Sprite

is that Sprite does not handle partitioning of the file tree since lookup for a file starts from root and proceeds downwards; whereas in VOFS longest prefix match is done on the entire path. VOFS allows virtual directories for keeping VOFS operational while at least one real object is in the tree. This feature and support for disconnected operation makes VOFS churn tolerant. Moreover, in Sprite every node exports resources under predefined prefixes and specific sub-directories in the tree while in VOFS a node can expose anywhere in the tree.

There exist peer-to-peer (P2P) file systems, e.g. OceanStore [13], which were developed as a file storage (data store) for file sharing. A typical P2P file system is used to store/retrieve files without support for neither POSIX file API access (i.e. the systems are not mountable), nor security. Grid file systems in contrast to P2P file sharing systems strongly require authentication and authorization to protect files from unauthorised access. VOFS allows the VO members to define and set VO security policies to be enforced by the VOFS security infrastructure.

Examples of Grid file systems include gLite file catalogs [14], Gfarm [15], and Distributed File Services, DFS [16]. The gLite *file catalogue service* [14] is used to maintain location information about files and their replicas. In contrast to VOFS, gLite catalogue service is centralized and is a single point of failure. The Gfarm file system [15] uses a virtual tree and virtual directories mapped to physical files by a metadata server, like the *centralized* solution described in [3]. Gfarm is designed to be very scalable; however, its metadata server can become a bottleneck and is a single point of failure since it is not replicated in contrast to VOFS. DFS [16] is a P2P file and storage system that can be integrated with a Grid security mechanism. DFS, in contrast to the presented VOFS, has no hierarchical namespace, but instead offers two P2P networks: one for storage space and one for names and metadata. DFS is implemented using FUSE [17] that limits its usage only to Linux, while the WebDAV-based VOFS can run on multiple (if not all) operating system, e.g. MS Windows, Linux, Mac OS X.

8 Performance

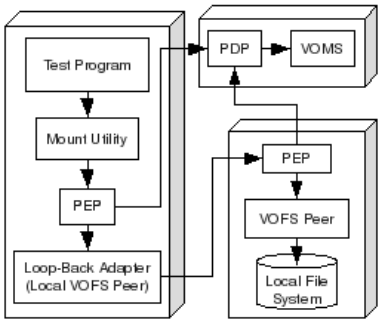

Fig. 4. Setup of performance test

We have made some experiments in order to evaluate performance, measured as file access time, of the VOFS prototype. Experiments include reading and writing files exposed to VOFS. The setup of the evaluation experiments is shown in Figure 4. The prototype under evaluation did not include encrypted communication described in Section 5.3. The nodes are PCs with 1.86 GHz Intel Centrino CPUs and 1 GB RAM on a dedicated 100 Mbps LAN.

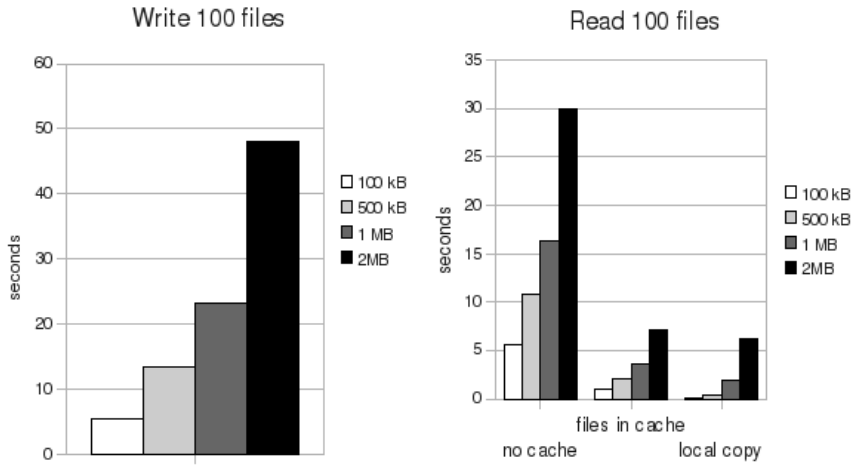

Fig. 5. Result of performance tests.

The read test reads 100 files from a remote peer and writes them to the local file system (outside VOFS). The file cache is big enough to contain all files. The results are presented in Figure 5. Figure also depicts timings for copying files within the local file system in order to compare performances of the local file system and VOFS. Bandwidth varies between 1.8 MBps for 100 kB files and 6.7 MBps for 2 MB files when no cache is used. Bandwidth when transferring smaller files is lower because of overhead takes proportionally more time. The overhead is mainly due to that the mount utility (davfs2) reads file properties before transferring files. The bandwidth is almost identical for 1 MB files and 2 MB files; this proves that in this case (for rather large files) overhead is negligible. It is also worth noting that copying 100 files from one local directory to another is about one second faster than copying of 100 files from the VOFS cache to a local directory. This means that VOFS introduces an extra overhead of 10 ms per file.

The write test copies 100 files from the local file system (outside VOFS) to a remote peer. Results of the write test are presented in Figure 5. Cache does not speed up performance since file content is written both to cache and the to remote peer.

Acknowledgments

Special thanks to Chen Xing (chenxing@kth.se) who implemented the first version of the VOMS and the gossip based protocol.

References

1. O.T. Anderson et al: Global namespace for files, IBM systems Journal Vol 43, No 4 (2004)
2. WebDAV Community, http://www.webdav.org/
3. Hamid Reza Mizani, Liang Zheng, Vladimir Vlassov, Konstantin Popov: Design and Implementation of Virtual Organization File System for Dynamic VOs. Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering - Workshops - Volume 00, pp 77-82 (2008)
4. Grid4All, EU grid research project, http://www.grid4all.eu/
5. davfs2, mount utility for WebDAV on Linux, http://dav.sourceforge.net/
6. Howard, John H: An Overview of the Andrew File System. Winter 1988 USENIX Conference Proceedings, pp. 23-26 (1988)
7. Rachid Guerraoui, Luís Rodrigues: Introduction to Reliable Distributed Programming. SpringerVerlag, Berlin Heidelberg (2006)
8. NetDrive, mount utility for WebDAV and FTP on MS Windows, http://www.netdrive.net/
9. Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marqus, Marc Shapiro, Pierre Sutra, Georgios Tsoukalas: Telex: Principled System Support for Write-Sharing in Collaborative Applications. Research Rapport, INRIA RR-6546 (2008)
10. Bo Lang, Ian Foster, Frank Siebenlist, Rachana Ananthakrishnan, Tim Freeman: A Multipolicy Authorization Framework for Grid Security. in Proceedings of the Fifth IEEE Symposium on Network Computing and Application, pp 269-272 (2006)
11. Von Welch, Ian Foster, Carl Kesselman, Olle Mulmo, Laura Pearlman, Steven Tuecke, Jarek Gawor, Sam Meder, Frank Siebenlist: X.509 Proxy Certificates for Dynamic Delegation. In Proceedings of the 3rd Annual PKI R&D Workshop (2004)
12. Brent Welch, John Ousterhout: Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System. Report No. UCB/CSD 56/261, Computer Science Division, University of California, Berkeley, California (1985)
13. Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz: Pond: the OceanStore Prototype. Proceedings of the 2nd USENIX Conference on File and Storage Technologies, San Francisco, pp 1-14 (2003)
14. gLite, middleware for grid computing, http://glite.web.cern.ch/glite/
15. Osamu Tatebe, Satoshi Sekiguchi, Youhei Morita, Noriyuki Soda, Satoshi Matsuoka: GFARM V2: A Grid File System that Supports High-Performance Distributed and Parallel Data Computing. Computing in High Energy Physics and Nuclear Physics, Interlaken, Switzerland, pp.1172 (2004)
16. Antony Chazapis, Georgios Tsoukalas, Georgios Verigakis, Kornilios Kourtis, Aristidis Sotiropoulos, Nectarios Koziris: Global-scale peer-to-peer file services with DFS. Grid Computing, 2007 8th IEEE/ACM International Conference on, pp 251-258 (2007)
17. FUSE: Filesystem in Userspace, http://fuse.sourceforge.net/