Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)
Thematic Priority 2: Information Society Technologies

Deliverable 3.2: VBS+DFS

Due date of deliverable: June, 2007.

Actual submission date: 20th June 2007.

Start date of project: 1 June 2006 Duration: 30 months

Organisation name of lead contractor for this deliverable: ICCS

Revision: Submitted 2008-06-20

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level

PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contents

List of Figures

List of Tables

Abbreviations used in this document

Abbreviation/acronym	Description
DFS	Distributed File Services
VBS	Virtual Block Store
MDDB	MetaData DataBase
CUID	Cryptographic User IDentity

Grid4All list of participants

Role	Part. #	Participant name	Part. short name	Country
CO	1	France Telecom	FT	FR
CR	2	Institut National de Recherche en Informatique en Automatique	INRIA	FR
CR	3	The Royal Institute of technology	KTH	SWE
CR	4	Swedish Institute of Computer Science	SICS	SWE
CR	5	Institute of Communication and Computer Systems	ICCS	GR
CR	6	University of Piraeus Research Center	UPRC	GR
CR	7	Universitat Politècnica de Catalunya	UPC	ES
CR	8	ANTARES Produccion & Distribution S.L.	ANTARES	ES

Preamble

This document is Deliverable 3.2, "Interface specification and initial running prototype of the Virtual Block Store".

The following persons contributed to this document:

Georgios Tsoukalas, ICCS
Antony Chazapis, ICCS
Georgios Verigakis, ICCS
Kornilios Kourtis, ICCS
Aristidis Sotiropoulos, ICCS
Nectarios Koziris, ICCS

1 Introduction

The present document contains the requirement analysis, design and implementation plan for DFS and VBS, undertaken by ICCS as part of the Grid4All 12th month deliverable D3.1, and a short report presenting the prototype implementation developed by ICCS for the 12th month Grid4All deliverable D3.2.

Task 3.1 defines VBS as a distributed block store on top of which DFS operates as a form of distributed filesystem.

VBS stands for Virtual Block Store. VBS is a software system that aggregates storage resources from internet peers into a single raw storage image that can be used by other systems, typically a file system, to allocate and access storage space.

DFS stands for Distributed File Services. DFS is a software system that hosts a file namespace on every peer. We use the term *file namespace* instead of *file system* because DFS stores only the hierarchical structure and metadata of files but not the file contents. Instead, pointers to VBS storage locations where the file contents are included with the metadata. DFS thus users the VBS layer for storing file contents. File namespaces are interlinked and a web of files is created over the internet — a unique approach to a distributed file system, that answers to many of the challenges of the Grid4All vision. The web of files does not possess a global root and peers choose their own entry point that leads to a different view.

VBS and DFS together provide a storage substrate for VOFS (VO-aware File System, Task3.2) and Semantic Store (Task3.3). Initially, the VBS and DFS were described as separate layers that would benefit from existing technologies put together to deliver the expected result. The initial ideas where substantially developed and this deliverable presents a unified architectural design that covers both DFS and VBS, since the two systems have a great deal to share in mechanism and functionality. Both need peering capabilities, authentication and authorisation mechanisms, and access to local resources.

Therefore, the design now specifies a unified VBS+DFS architecture, which will be referred to as the ***DFS Architecture***, throughout this document. The DFS architecture features generic mechanisms that create peers capable of communication, access local storage resources and implement protocols for peer-to-peer services. The design includes two protocols. One is for namespace and metadata serving, the MDDB protocol, and the other is for storage serving, the VBS protocol. The VBS layer is thus included in the overall DFS architecture.

Because of the unified VBS+DFS architecture, the D3.2 deliverable prototype implementation includes much of the DFS functionality, in addition to the VBS functionality that was initially planned.

The most prominent features of the DFS architecture are:

- Filesystem servers and clients as peers on the Internet
- Peer-to-peer filesystem access through symbolic linking between namespaces
- Explicit separation of file hierarchy and metadata against file data storage
- Operation while disconnected from the network
- Integration (mounting) with the local OS filesystem hierarchy
- Global index of peers over a structured peer-to-peer network
- Authentication and authorisation by public-private key cryptography

In summary, the Grid4All vision poses mainly two scenarios as challenges to the storage substrate:

- Users must be able to find and allocate distributed resources to enable application functionality, because their own resources don't suffice.
- Users must be able to share distributed resources to enable collaboration among them.

For both these scenarios, users create task-oriented Virtual Organisations (VOs), in order to manage the tasks of finding and sharing resources. The DFS architecture provides support in that respect both to the VOFS, that interfaces with the VOs, and the Semantic Store, that interfaces with the applications.

The DFS architecture design philosophy can be expressed by two important principles:

- Total resource control for resource owners.

  This means that resource owners can enforce their own policy for access to their resources, no matter prior negotiation or policy arrangements from third parties.
- Semantic and functional independence among peers and views of the peer network.

  There does not exist a single global view of the peer network and its resources. Views and operations may be conflicting among the peers without damaging core mechanisms; peers can survive any conflict or remote failure falling back to their local view. The general term *view* refers to the state of remote objects as it is registered locally at a particular peer.

The structure of this document is as follows:

Section 2 presents a wide range of storage systems that are relevant to the DFS and VBS design and implementation. The design incorporates many of the techniques, concepts and experience of presented state of the art.

Section 3 discusses the requirements that the general Grid4All scenarios have on the DFS architecture as a storage substrate.

Section 4 presents the DFS architecture, including the VBS layer.

Section 5 presents the DFS+VBS prototype implementation.

Section 6 describes detailed usage scenarios.

Section 7 details the connections of this work with other tasks of the project.

2 State of the Art

2.1 Block Storage

2.1.1 iSCSI

iSCSI ([06]) defines a transfer protocol for SCSI over typical TCP/IP networks. iSCSI was developed as a cost effective solution, because unlike other alternatives such as Fibre Channel ([08]) it can be used over ubiquitous TCP/IP networks. iSCSI initiators can be implemented completely in software using common NICs. Modern Operating Systems such as Linux and windows support this kind of functionality. In practice, however, in order to achieve better performance a part of the desired functionality can be implemented in hardware and specifically in the network card. Examples are TCP Offload Engines (TOE) which implement the TCP/IP protocol in the NIC and iSCSI Host Bus Adapters (HBAs) which implement the full iSCSI protocol and expose themselves as a SCSI HBA to the operating system.

2.1.2 ATA over Ethernet

ATA over Ethernet (AoE) ([09]) is a lightweight protocol similar to iSCSI, designed for accessing ATA storage devices over Ethernet networks. Unlike iSCSI, AoE does not use the TCP/IP protocol stack, which enables the protocol to achieve good performance without the use of expensive adapters. The downside of this design decision is that the AoE protocol can apparently be used only in Ethernet LANs. There is also a protocol that supports transportation of SCSI commands using raw Ethernet packets which is called HyperSCSI ([12]).

Network Block Device (NBD) is a block device driver for the Linux operating system. NBD implements a virtual block device that forwards all requests to a userspace server over a TCP/IP network. The server handles the requests and sends the result back to the client. NBD, allows only a client at a time for each storage device exported by the server. There are various expanded versions of NBD that provide additional functionality. ENBD, which stands for Enhanced Network Block Device, provides additional features such as: multichannel communication, internal failover and automatic balancing between the channels, encryption and authentication. As noted in the webpage of the ENBD project: "*The intended use for ENBD in particular is for RAID over the net*". Relate work includes [10] which discusses implementation issues in order to use nbd for data replication (RAID1) and [11] which evaluates performance of a distributed RAID5 implementation based on NBD and concludes that in most cases performance of distributed RAID is noticeable better than that of a single disk system. Additionally, there is also GNBD, which is part of the GFS Filesystem and stands for GFS Network Block Device. The difference between GNBD and NBD is that GNBD supports multiple clients on a server, all accessing the same exported storage resource.

2.1.3 Petal

Petal [14] was a research project at Compaq's System Research Centre. Petal consists of a pool of distributed storage servers that cooperatively implement a single, block-level storage system. The system provide high-available large abstract containers, called virtual disks. Global state information is maintained among the server nodes by an algorithm which is based on Leslie Lamport's Paxos, or "part-time parliament" algorithm [15] for implementing distributed, replicated state machines. The petal system provides a mapping from `<virtual-disk-id,offset>` to `<server-id,disk-id,disk-offset>`, based on a 3-level mapping table structure, of which the first 2-levels must be global (replicated to all servers). Petal supports a redundancy technique called *Chained Declustering* [16] which is similar to RAID, incremental reconfiguration and automated backup and restore capabilities.

2.1.4 NASD

Network Attached Secure Disks (NASD) as presented in [13] was a research project at Carnegie Mellon University which aimed to mitigate the scalability problems of typical NAS architectures such as NFS. NASD eliminates the unique server bottleneck by enabling storage devices to transfer data directly to clients. Storage devices export variable length objects instead of fixed-sized blocks as an interface. The higher-level semantics (such security, metadata) are handled by the central file manager.

2.1.5 FAB

FAB (Federated Array of Bricks) as presented in [20] is a distributed disk system built on *bricks*, which are commodity systems containing disks, CPU, NVRAM, and NICs. The interface provided by bricks, to the storage clients is [06]. Clients send an iSCSI request to a brick, which acts as a coordinator and forwards the request to the appropriate storage bricks based on the technique used. FABs support specific replication and erasure-coding algorithms [21], which are both based on *voting*: Each request makes progress only after receiving replies from a (random) quorum[1] of bricks. As in the case of the Petal global metadata are kept on all bricks using Paxos [15]. A group of bricks, called *seggroup* is mapped to the logical offset of a device at *segment* granularity. A large *segment* size value (256MB) is chosen to reduce the global metadata management overhead. The local mapping on each brick has smaller (8MB) granularity (*page*). Timestamps are used for the various consistency protocols.

There are also various other attempts similar to build highly-efficient and highly-available storage systems based on *bricks*. An example is self-* storage [19], which explores the design and implementation of self-organising, self-configuring, self-tuning, self-healing, self-managing systems storage systems based on *bricks*, borrowing ideas from AI and corporate theory [18] . The target of this project is to *“develop and deploy a large-scale (1 PB) storage constellation, called Ursa Major, with capacity provided to research groups (e.g., in data mining and scientific visualisation) around Carnegie Mellon who rely on large quantities of storage for their work*. There is also an IBM project called Intelligent Bricks (Intelligent Bricks Hardware, Intelligent Bricks Software, [22]).

2.1.6 Unclassified

[17, Semantically-Smart Disk Systems] presents an approach where smart disks operate based on some knowledge about the upper layer (gray-box approach), which in most cases is the filesystem. This knowledge is acquired by a fingerprinting tool that automatically discovers file-system layout through probes and observations.

2.2 Globally Distributed Storage Systems

Globally distributed storage systems share a lot of properties and design issues with LAN Storage Systems, but there are also some inherent differences among them:

- On globally distributed storage systems connectivity of a large number of nodes is intermittent.

- Network Latencies can be very high, and one can't assume a high speed network interconnection among nodes.
- When data distribution is global, there are more security issues that need to be resolved.

technology	type	medium	redundancy	encryption	security	block size
iSCSI	client / server	TCP	NO	NO	CHAP	arbitrary
ATA over Ethernet	client / server	Ethernet	NO	NO	NO	fixed
NBD	client / server	TCP	NO	NO	NO	fixed
Petal*	distributed	ATM	Chained Declustering	NO	NO	fixed
NASD	distributed	UDP	RAID-like	YES	Capabilities	variable
FAB*	distributed	iSCSI	replicas / erasure-coding	NO	NO	fixed

* Requires NVRAM

Table 1: Qualitative comparison of networked block storage systems

2.3 Filesystems

2.3.1 Log Structured Filesystems

A log structured filesystem (LFS) [34] treats the storage device (e.g. disk) as an append-only log. New data (either by creation of new files, or by modification of existent) and their corresponding metadata are batched together and written to the end of the log in large sequential transfers. LFS performs well when writing small files and translates temporal data locality to spatial storage locality: data that where written at the same time are stored together and can be retrieved efficiently. Typical filesystems, such as [33], translate logical data locality to spatial storage locality by storing together data that are close in the filesystem namespace. LFS provides a segment cleaner, which is responsible for garbage-collecting the free space and coalescing it into large contiguous regions. LFS also is also able to recover from a crash more quickly than most file systems, because its append-only nature ensures that only the tail of the log needs to be checked for inconsistencies. LFS has two features that render it well-suited for RAID-like storage devices: It operates with large transfers (no partial stripe writes) and easy recovery.

2.4 Network File Systems

Network File Systems, are the most popular implementation of the NAS architecture. The server exports a local filesystem to various clients via a specific protocol. The client uses this protocol to provide users with a file system interface for the remote storage that is exported by the server.

2.4.1 NFS

The most common Network File System in UNIX and UNIX-like operating systems is NFS, which was originally developed by Sun. There are various versions of NFS ([03], [04], [05]). Versions 2 and 3 use UDP as a transfer protocol, although it is possible to use TCP from version 3. Version 3 includes various performance modifications such as support for asynchronous writes on the server, security enhancements by allowing support for an access check to be done on the server and extensions to allow 64 bit file sizes and offsets. Version 4 implements a stateful NFS protocol, which allows file locking to be integrated in the protocol and not implemented outside. Additionally, Version 4 of the NFS protocols includes support for strong security, client caching, compound operations and internationalisation.

Another wide-spread network filesystem is SMB (Server Message Block), which was originally developed by IBM. SMB was later extended by Microsoft which renamed into CIFS (Common Internet File System)

2.5 Shared Disk File Systems

Shared disk filesystems are special-purpose filesystems for nodes that have access to a shared storage pool. When multiple nodes are accessing the shared storage a common filesystem won't suffice, because consistency can't be maintained.

2.5.1 Global File System

Global File System (GFS) ([23] , [24]) is the most popular shared-disk filesystem. Originally, GFS was developed for the IRIX Operating System, although now it is most commonly used in Linux systems. GFS is fully distributed, in the sense that there aren't special nodes, all operations can be equally performed by every node. All data and meta-data of the filesystems are equally distributed on all storage devices.

Initially, the synchronisation between the nodes of the filesystem was implemented with special SCSI commands during the writing of metadata information to the storage devices. For devices that did not support those operations, an additional layer was added, which supported the synchronisation of nodes via a central server. The special SCSI commands were not supported by many vendors, so the centralised synchronisation scheme prevailed. This posed a performance and reliability problem so Distributed Locking Mechanisms (DLM) were developed, to protect the data and meta-data of the filesystem.

2.6 Distributed File Systems

Distributed File Systems are filesystems that provide a unified filesystem view, supporting multiple clients and possibly multiple storage servers. The main difference with Network File Systems is that the latter support only one storage server.

2.6.1 AFS

Andrew File System (AFS) is a distributed file system developed initially by Carnegie Mellon University. AFS clients use a persistent (on disk) cache that caches both file and directory data. In order to support this caching scheme, servers notify clients with a callback to discard data, if data have changed. These changes are propagated when a file is closed and not when is written, as in typical Unix systems. The servers provide a common namespace for all clients, so a file is always identified by its location. AFS provides improved security features such as the use of the kerberos protocol for user authentication and an ACL based protection scheme. Read-only replication of data between servers is also possible. AFS finally supports wide-area operation.

Coda [25] is a filesystem that was based on AFS. Coda supports disconnected operation via operation logging and data replication via a read one/write all model and versioning of individual replicas. InterMezzo filesystem was inspired by Coda, and was coded from scratch, targeting in much smaller implementation size.

2.6.2 Zebra

The Zebra stripped network filesystem [35] is a filesystem that stripes file data across its servers, in order to provide a scalable storage system. Zebra is build on two basic technologies: RAID and Log Structured Filesystems. The Zebra filesystem uses log-based stripping (instead of file-based stripping) and as LFS uses the logging approach at the interface between a file server and its disks, Zebra uses the logging approach at the interface between a client and its servers. Zebra performs the parity computations on the clients. One of the most important features of Zebra is that it updates file blocks by writing new copies, rather than updating the existing copies. This approach has several advantages: It allows the clients to batch together blocks from different files and write them to the servers in a single transfer and several clients simultaneously modify data without synchronisation. To allow clients not only to store, but also to share data a centralised file manager is provided, which manages the metadata for locating data and for consistency issues. Another scalability problem for Zebra is that Zebra, like LFS, relies on a single cleaner to create empty segments.

2.6.3 xFS

xFS [37] is a serverless network filesystem in which all nodes cooperate as peers to provide all file system services. xFS distributes control processing across nodes on a file granularity, distributes storage data using a software RAID mechanism based on log-based network striping similar to that implemented in Zebra and implements *cooperative caching*, where portions of client memory are used as a large global file cache and data requests can be satisfied by data cached on other machines [36]. xFS is one of the first fully-distributed peer-to-peer systems, although its intended use was for fast-access LANs and not for large global networks.

For each file there exists a *manager*, which controls two sets of information about it, cache consistency state and disk location. There are four mappings data structures for locating distributed data and metadata in a xFS system: the *manager map* maps files to managers and its globally distributed among nodes, the *imap* maps file data to on-disk log locations and its located on the manager of a specific file, the *file directories* map file names to inodes and *stripe group map*, which maps disk log address to list of storage servers and its globally replicated. xFS, also distributes cleaning operations among nodes of the system.

2.6.4 Lustre

Lustre ([26] , [27]) is a POSIX compliant distributed filesystem that scales to tens of thousands of nodes and petabytes of storage. Lustre is open source and its name its an amalgam of the terms “Linux” and “Clusters”. There are three different entities in a Lustre environment:

- **Clients** that use the storage space provided by the filesystem
- **Metadata Servers** (MDSs) which hold the metadata for the filesystem (directory layout, permissions and extended attributes)

- **Object Storage Targets** (OSTs) which are responsible for the storage and transfer of actual file data.

Lustre is a system that is designed around object protocols, where storage devices provide an object interface instead of a block address space. This technique allows for internal metadata to be handled on OSTs and thus provides better scalability. There are also a number of other approaches that utilise object storage such as: [13], [28], [29] . Lustre's network layers uses a simple message passing interface called *Portals* [30] and supports a large number of modern interconnection networks such as Quadrics, Myrinet, Infiniband [07] and TCP/IP. Lustre uses it's own Distributed Lock Manager (DLM), which among other things supports *intent locking*, a way for allowing the lock manager to choose between different modes depending on its view of resource contention.

2.6.5 PVFS

PVFS (Parallel Virtual File System) [31] is a distributed filesystem for clusters and its primary goal is to provide high-speed access to file data for parallel scientific applications. After the release of the first version, the second version of PVFS [32] is now developed. A basic design decision in PVFS2 is that it does not provide POSIX semantics, but defines a more relaxed consistency scheme. PVFS2 protocol is stateless and locks not are used as a part of client - server interaction. PVFS2 provides two different interfaces for applications: a typical UNIX I/O interface and an MPI-IO interface, to better leverage the capabilities of MPI-IO to perform efficient access in a PVFS2 system. Other features of PVFS2 include support for distributed metadata, data and metadata redundancy, tunable filesystem semantics and flexible and extensible data distribution modules.

2.6.6 Eliot

Eliot [39] is a distributed filesystem over a peer-to-peer network. File data are kept in an immutable content-hashed peer-to-peer block store while the rest of the filesystem is kept in replicated metadata servers. With immutable hashed file contents, data become easily cached and fakes are easily detected. On the other hand, having a centralised (although replicated) metadata server, the filesystem gains performance and manageability. Mass queries and operations can be applied at once, while the administrators of the servers can enforce strong security control. The idea that a centralised (or, more centralised) metadata server boosts performance and manageability can be found in the design of the Lustre high-performance cluster filesystem too [27]. Eliot implements open-close consistency like AFS.

3 Requirements analysis

Grid4All's main objective is to produce the necessary infrastructure that will enable domestic users, non-profit organisations such as schools, and small enterprises to participate in a massive resource sharing network over the Internet. The vision of the project is a "democratic" Grid where inexpensive resources are cooperatively pooled in a dynamic and scalable fashion. Even small-scale users that do not have the necessary computing and storage assets to participate in current Grid deployments should be able to contribute their resources and in turn be able to utilise the unified substrate. All services implemented in such an environment face great challenges regarding security, support for multiple administrative and management authorities, on-demand resource allocation, heterogeneity and fault-tolerance.

Most relevant issues have already been addressed in the context of peer-to-peer data sharing and resource location networks — most, except administration and authority, which contradict the purpose the majority of such systems have been designed for. In classic peer-to-peer overlay designs, participants agree on the protocol to be used, which in turn enforces the policy of resource allocation and usage. Peers donate their storage resources to a common pool which is then managed by the network itself. Of course, one can refuse to implement the complete protocol semantics or process selective commands in an arbitrary way, although the corresponding results of such behaviour are ambiguous. On the other hand, the design proposed in this document takes into account a peer-to-peer environment where:

- Peers have direct control of their resources. Each peer may administer its own storage and file objects and perform operations on them independently of their location and usage in the network.
- Peers have control of how their resources are used. Each peer may authorise specific peers to certain actions. Also, each peer may define its own sharing policy.

  The DFS architecture assumes that users who own resources, also run peers that provide access to those resources. Local control of peers makes policy enforcement possible while cryptographic identification of each peer and resource allows peers to specify access policies. Therefore, as long as communication among peers is authenticated, peers have total control over their resources. For details, see Section 4.1.
- Peers should be able to allocate and use resources they do not physically possess. This can be achieved either by pooling of resources or sharing, as long as the process complies with the previous requirements.

  Details about pooling storage resources and allocating on demand can be found in Sections 4.5.11, 5.6, 7.2
- All actions should be accountable. Every transaction in the network should be traceable to a named peer, resource or combination of two. See Section 4.1
- The network's capacity should grow as more nodes join it, in a typical peer-to-peer fashion. Moreover, well connected and well resourced nodes should be exploited when needed and if they allow so.

  Concerning files, this is largely a matter of policy and content and cannot be addressed at the level of mechanism much beyond offering peer-to-peer facilities. However, raw storage can be pooled in peer-to-peer networks, thus exploitng the distributed resources. VBS can implemented using a such a peer to peer network. See Section 5.6 for a presentation of a prototype implementation and Section 6.3 for a relevant scenario.

Moreover, the Grid environment we target has imposed special requirements, including:

- Shared namespaces. In addition to sharing file contents, participants should be able to agree on common collections or clusters of files. This is traditionally achieved through distributed filesystem designs where numerous peers agree on a common namespace of data. The storage subsystem should allow equal functionality, additionally supporting the ad-hoc creation and management of multiple such views.

  The DFS architecture provides this functionality with its filesystem to filesystem links (*metalinks*, see section 4.2.1). With metalinks, shared directories become shared namespaces since peers can logically link their resources there, while maintaining the control of their resources. For a relevant scenario see section 6.1

- Support for multiple storage types. As we presume cooperation among new and already deployed file services, we should provide mechanisms for merging existing data exported via GridFTP, FTP, HTTP, etc. into the same distributed namespace and allow seamless access to objects disregarding the transfer protocol or location.

- Support for special file types. Data contained in files may have special semantics, and as so require or support special operations beyond access, move, copy, delete, etc. For example log files may provide special mechanisms to append entries or files storing experimental results from scientific measurements may contain special metadata.

- Support for disconnected and offline operations. Network failures and mobility of users are expected to often make peers unreachable or disconnected from the main network. Users must be able to survive this without unnecessary loss of functionality or data.

4 DFS+VBS Architecture

4.1 Generals

The DFS architecture architecture (DFS) enables distributed access and management of files and storage resources in a peer-to-peer environment while retaining full local control and allowing accountability of corresponding actions. In essence, it is a peer-to-peer service-oriented architecture for distributed file operations that allows us to synthesise the diversity, scale and dynamicity of a typical peer-to-peer network with the control and accounting features. DFS allows users of the system to administer file metadata and actual data as two distinct entities, independently of each other.

Users of the system create and control peers that participate in a network of namespaces and storage. Through their peers, users are enabled in

- *Exporting files and storage to other users.*
- *Creation of shared namespaces where each user can export existing files under whatever name they choose.*
- *Creation and management of shared files.*
- *Aggregation of multiple users' files under a single namespace. This enables distributed resources to be authenticated as "official" under the authority of this single namespace.*
- *Disconnected operation on their critical data.*

The DFS Architecture is based on the following principles:

- *Cryptographic User IDentification* ***CUID***

  DFS mechanisms identify users by a Cryptographic User IDentification (CUID) token. A cryptographic certificate is retrievable from the CUID, which will authenticate the CUID. Users are named, authenticated, authorised, accounted, attributed an action or associated with each other and external entities, all with the use of CUIDs.

  CUIDs may be created arbitrarily by users, but policy may require certification from authorities or other associations, such as inclusion in white lists or high scores in peer-to-peer reputation systems.

- *Any* ***resource*** *is identified by a single* ***authority****, a single CUID*

  Any resource and policy concerning resources is identified by and submitted to the unconditional authority of a unique DFS CUID. This has two main benefits. This makes mechanisms resistant to negotiation or reconciliation failures, since the authority can always provide a sound and enforceable resolution to any matter.

  The simplest way to handle conflicting operations on resources is to serialise all operations before execution at the site of the single authority. In the cases this best effort consistency is not sufficient, the applications must use their own mechanism to synchronise. External mechanism can be supported by special privitive operations if needed.

- *Any* ***action*** *upon a resource is attributable to a single CUID*

  The agent of any action concerning resources is ultimately identified by a single CUID. This facilitates the accountability of users for their actions and simplifies resource access control.

Combining the two previous principles we can see that any action upon a resource is essentially a transaction between two CUIDs, an agent and an authority. The agent must acquire the permission of the authority in order to successfully perform the action.

4.2 Resources

The proposed DFS architecture explicitly distinguishes between two kinds of resources, the filesystem data and the actual storage space. The first kind of resource will be called the FileSystem resource and it refers to the usage and manipulation of the information stored in a filesystem. The second kind will be called the Storage resource.

4.2.1 Filesystem Resource

The ***filesystem resource*** refers to a namespace and its data, as in computer filesystems. It should be noted that only the information content belongs to the resources — storage space for file data belongs to the storage resource, defined next. Of course, the filesystem metadata require storage space too. This storage space is considered part of the filesystem resource.

Follows a list with the basic features of filesystem resources.

- Three constituents: ***path to file***, ***file metadata***, ***file data***

  The filesystem resource constitutes of three parts: path, metadata and data.

  The path represents a file, including directories or other special file types. The path is important because it names the file data. For example, the list of users that can login in a unix system is maintained in the file under the path `/etc/passwd` . All programs that implement relevant functionality, trust that this path points to authoritative data. The ability and act of making data accessible under a certain name is essential part of the filesystem resource. The resulting hierarchy of names will be often referred to as the ***filesystem structure***.

  The metadata encode the policy according to which both filesystem structure and file data can be manipulated. A simple model is adopted: there is a list of operations and there are Access Controls Lists (ACLs) for mapping CUIDs to their permissible operations. It is noted that permission from the filesystem is not in principle capable to grant a user access to the data because the actual data storage belongs to a different resource. This filesystem to storage delegation is discussed later on.

  Data belong to the filesystem resource as a ***storage pointer***. The actual data is not (in principle) included in the filesystem resource but is instead linked to. This is dictated by the principle of explicit filesystem and storage resource separation. It offers flexibility, as the producer and owner of information need not logically be the host. In fact it is often the case that more storage is needed than available or, conversely, more storage is available than needed. The interaction between the resources is discussed after the user roles are defined.

- ***URI identification***. A file as a resource is identified by a URI of the form:

  `dfs://Owner:fs/Path`

  `Owner` is the identification of the owner of the filesystem (see section Roles Definition). `Path` is the path within the filesystem. The `dfs://` prefix stands for the entire protocol family within a DFS network, while the service specification `:fs` denotes the specific target service.

- ***Metalinks*** from namespace to namespace

  A special type of file exists, the ***metalink***, that points to an arbitrary filesystem URI. In that way, it links its path to the pointed URI providing the ability to link filesystems arbitrarily together, forming simple or more complex structures. The parent filesystem cannot control the structure of its descendants. This feature is very useful in the formation of peer to peer filesystem structures that combine resources from different CUIDs to be shared. Metalinks in the DFS network can be considered as an extension of the traditional filesystem symbolic links.

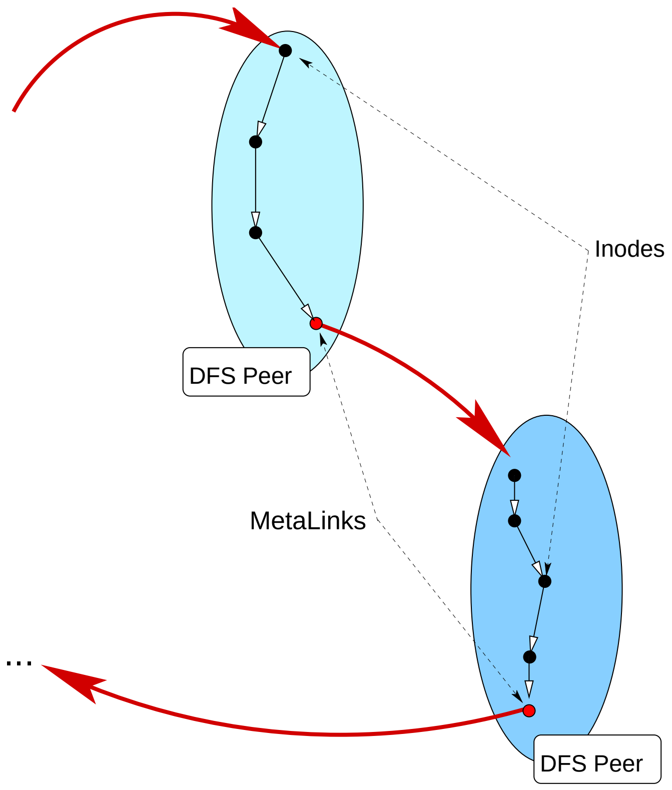

Figure 1: Filesystem Metalinks

4.2.2 Storage Resource

The storage resource refers to the actual storage of content, such as file data. Filesystem resources are used to associate data with users and policies whereas storage resources are consumed to store data and make them available for retrieval over the network. ***Data blocks*** are the units of storage allocation and are identified by URIs of the form:

```
dfs://<provider>:storage.<protocol>/<handle>
```

The URI includes the providing CUID, the specific protocol that may be used to access data, and a handle that is used by the provider to identifies the data block. This implies that storage is allocated.

In the DFS Network, the size of data blocks is not assumed to be uniform or constant. This allows data blocks to closely resemble regular files. Therefore, clients who only need storage may directly use Storage Resources, if the storage provider supports mutable and resizeable data blocks.

4.3 Roles

According the principle, each action upon a resource is essentially a transaction between an authority CUID who owns the resource and an agent CUID who performs the action. The authority has to give its permission for the agent to complete the action. Taking into account the given resource definition, three roles for users are defined to represent the authority and the agent for each action. These roles are shaped such as to address the needs for DFS resource access and management. The roles provide an abstract framework that guides the DFS software component design and implementation, as well as the interface of the whole DFS subsystem to the other layers.

It must be stressed that any CUID can play all of these roles.

4.3.1 File Owner

Each CUID in the network is entitled to its own namespace and files. Conceptually, file hierarchies are not physical resources in the sense that they are just information. Owners can either authoritatively serve their file resources through authenticated communication channels or cryptographically protect their content when hosted at untrusted sites.

Of course, a hierarchy of files as information, has to be placed in physical storage. The storage space for file hierarchy and metadata are served by the Owner. The file contents are served by providers of storage resources. It is the Owner's responsibility to find providers of storage for its files.

4.3.2 Storage Provider

As with files, each CUID can own storage resources. Storage Providers can serve requests for storage allocation and access after they authenticate their peers and authorise their requests. Providers can contribute storage to other Providers, so a big part of their role is to make contact with others and establish a peer-to-peer storage network. Aggregation of storage essentially means that a Provider can present for storage management a single virtual image of storage space that is arbitrarily distributed and has been registered with it as contributions.

The storage contribution and aggregation patterns are policy that is prescribed by the users in control of the CUID. It is expected that most Owners will utilise their own storage resources for their files.

Providers can serve storage from any underlying storage system they need, as long as they conform to the specific protocols. This makes data managemet flexible. Of course, the choice of lower level storage can limit the available operations and services. For example, an alternative storage device is planned as a peer-to-peer block store where each peer can provide storage in a DFS network.

4.3.3 Resource Consumer

The third role is that of the Consumer of the resources. This is straightforward, as each peer can contact any other to request access to any resource. Consumers are expected to be controlled by client applications.

4.4 Component Definition

The software component architecture of the DFS describes how the abstract roles framework is to be implemented in software. The first concern is the definition of the interface among the various such components. Secondly, implementation alternatives for each component are presented.

The main goal for each component is to employ existing technologies and software, coordinated by a thin layer that interfaces with the other components according to the architecture specification.

4.4.1 DFS Architectural Objects

Resources, data and other conceptual entities that exist in the DFS Architecture, need concrete representation in storage and communications protocols. For this reason, all the software components are required to represent their data at interface and protocol context with a well defined set of ***architectural objects***. Each architectural object has a predefined set of possible attributes. Each attribute has a name and a type. The type must be another architectural object type. This hierarchy ends with basic types for numbers, text, etc.

The architectural objects have two forms of representation. One abstract, as object constructs of the programming language and one concrete, a serialised string representation. For some objects, such as the DFS URI, the serialised form is well known as part of the user interface and directly appear in the input or output of the software.

Object instances may include only a subset of their possible attributes when they are used to refer only to those attributes. In this way, for example, one can overwrite a single attribute of an object without having to retrieve it first. In more complicated uses, several attributes can be implied by the context and therefore partial reference to the object contents is necessary.

The architectural objects are part of the lowest level interface to the DFS. Peers that can handle these objects can participate in a DFS Network irrespective of their specific software origin. However, standard clients will be provided that will export a higher-level POSIX-like interface for applications that do not need to extend existing DFS mechanisms.

Follows a description of the higher level object types. Attributes that are not described are labeled with a type *opaque*. Object instances may not include all attributes when they are used to refer to some stored object, or when some fields are unknown or implied elsewhere. This way, one can selectively refer to specific object fields when issuing a store, retrieve or search operation.

DFS URI The DFS URI is a string that is comprised of some mandatory and some optional components with proper separators.

```
dfs://<user>:<service>.<protocol>/<path>
```

Attribute	Description
`<user>`	The key that identifies the Owner of the resource. This attribute is of type DFS CUID.
`<service>`	Type DFS Service, together with the `<protocol>` attribute. The identifier of the resource type. There are two types of resources, filesystem resources with an identifier `fs` and storage resources with an identifier `storage`. If a service is not specified, then the default is `fs`
`<protocol>`	This field is used as a parameter to the service specification Specifies service specific parameters
`<path>`	This is the local identifier of the resource. For filesystem resources this path is parsed and traversed. For storage resources it is used as an internal handle for an object, such as a filename, or as an identifying string for a specific block, such as a block's content hash in a P2P block store.

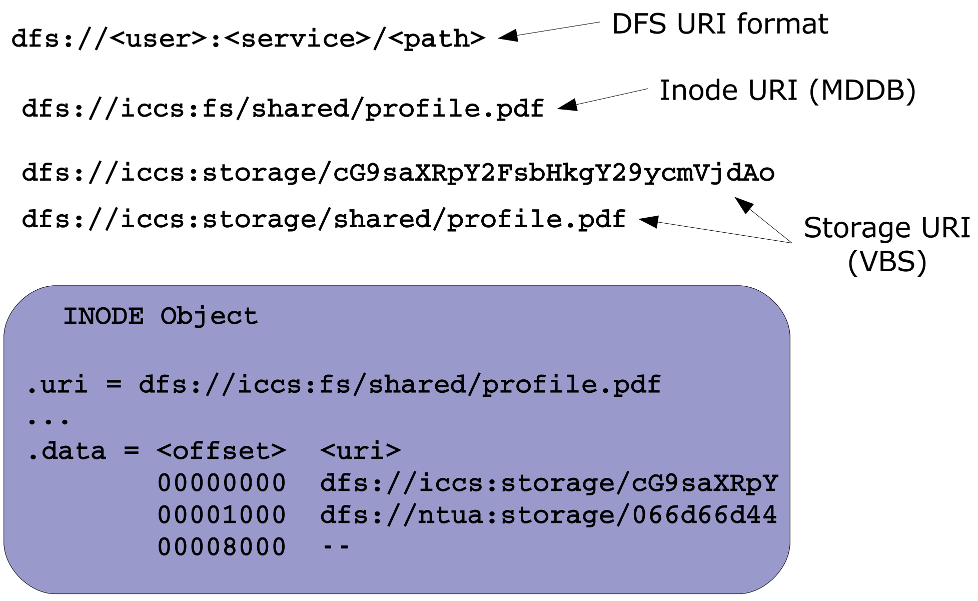
dfs://<user>:<service>/<path>
DFS URI format
dfs://iccs:fs/shared/profile.pdf
Inode URI (MDDB)
dfs://iccs:storage/cG9saXRpY2FsbHkgY29ycmVjdAo
dfs://iccs:storage/shared/profile.pdf
Storage URI (VBS)
INODE Object
.uri = dfs://iccs:fs/shared/profile.pdf
...
.data = <offset> <uri>
00000000 dfs://iccs:storage/cG9saXRpY
00001000 dfs://ntua:storage/066d66d44
00008000 --

Figure 2: Schematic representation of a URI object

DFS CUID A ***DFS CUID*** object is actually a cryptographic certificate (or an alias for it that can be used to verify the authenticity of the users' requests and data. Text aliases can be used in order make handling more convenient.

DFS Service The DFS ***Service*** object is a text identifier of a protocol and optionally arguments to the protocol. For example, `fs` identifies a filesystem service while `store.p2p` identifies a storage service and specifies a particular protocol '`p2p`' for its access.

DFS Path The DFS ***Path*** object resembles a conventional path string. It can be broken to components that will be traversed successively, as part of a DFS filesystem URI, or can be used as a handle to an internal storage object (e.g. a filename) as part of a DFS storage URI.

Container A ***Container*** is a special object that can contain an arbitrary collection of named DFS Objects. The objects are storable and retrievable by their name. Containers can be used as object holders in protocol messages or other objects with dynamic structure, such as Inodes.

DFS Inode `Inode:{<uri>, <metadata>, <data>}`

Attribute	Description
`<uri>`	Type DFS URI. This is the identifying URI of the Inode
`<metadata>`	*Opaque*. Represents the files' metadata
`<data>`	Of type Container. The container represents the file's data. According to the inode type, the container may contain objects of different types.

The ***DFS Inode*** object is the central object of the software interface to the file services. Its name is historical; i-node stands for internal node. Filesystems are trees with file data in the leaves and internal nodes that hierarchically contain them. The Inode object represents a DFS FileSystem Resource. That is to any given filesystem URI (i.e. one with an '`fs`' service) corresponds exactly one Inode object that describes this resource. The URI under which an Inode was created by its Owner is reflected in the `uri` field. Note that the Owner of the inode is included in the `uri` field. The `metadata` field contains information about the inode type (file, directory, etc), attributes, access control lists (ACL), etc. The `data` field contains objects referenced by the Inode's metadata, such as pointers to data storage, directory entries, access list objects, or other special objects.

Special Inode types Apart from regular data files and directories, special inode types can be defined. Because of the flexibility that the Container offers, Inode types can be made opaque to the file service's servers and Inode objects be interpreted by clients according to arbitrary protocols.

Metalinks An important special Inode type in the DFS is the ***metalink***. A metalink resembles a traditional filesystem symbolic link in that it specifies another file as the target file to be retrieved instead of itself. But the metalink gives a full FileSystem URI specification for its target, thus linking directly within (potentially) another filesystem in the DFS Network, served by another peer. With the use of metalinks, the DFS peer network can structure its namespace in arbitrary ways, much like the world wide web does.

DFS Message `Message:{<verb>, <arglist>, <container>}`

Attribute	Description
`<verb>`	A text string of protocol specific semantics
`<arglist>`	A list of text strings as arguments to the verb
`<container>`	A container of objects referenced by the verb or its arguments

DFS Messages are exchanged among DFS Peers to transfer request and reply data according to the various DFS Protocols. Each message has a ***verb*** field, that corresponds to a request or a reply. For example, a verb `'RETR'` may represent a request for a file from a file service. As another example, a verb `'ACK'` may represent an acknowledgement and successful reply of such a request. Additional text arguments may follow. For example,

- `'RETR 717 /opt/example/file'`
- `'ERR 717 404, File does not exist'`

The container of the message holds whatever architectural objects are referenced by the arguments and are needed to be transferred as part of the request or reply.

DFS protocols specify their own verbs and message contents, but they all are required to use this common message format.

DFS Protocols The DFS Message structure suggests a generic layout for protocols. A protocol message includes a `VERB` that defines the request, along with its `arguments`. The `VERB` and the `arguments` can implicitly or explicitly refer by name to objects stored in the message Container. This provides a simple but powerful mechanism for easy protocol creation. To create or extend a protocol, one has to define the Verbs, and implement their handling and any additional object types. The common message format permits clients to ignore protocols or protocol extensions they do not support or even to partially or abstractly parse messages.

All protocols use some common verbs with well known semantics so that even if the contents of the messages exchanged are not understood, the communication is always manageable. Examples of common verbs are `ACK, ERR` that report success or failure, and those for session management and authorisation.

4.4.2 DFS Peers

DFS Peers are software components that operate as communication endpoints for a DFS Protocol. They are identified as peers by a URI that includes their owner and their service identification.

Each role in a DFS Network is implemented by a peer to peer software component. The respective software components for the roles of Owner, Provider and Consumer are the MetaData DataBase (MDDB), the Storage Pool and the DFS Client. The Storage Pool incorporates the Virtual Block Store layer that manages storage allocation.

4.4.3 MetaData DataBase (MDDB)

MetaData DataBase (MDDB) peers implement the filesystem resource. MDDBs store namespace hierarchy and file metadata including authorisation information into a local database. Part of the file metadata are the storage pointers that point to the storage peers that contain file data. Metalinks from namespace to namespace link MDDBs together creating a web of files. This functionality is essential as it allows the creation, management and authentication of shared workspaces where distributed resources can be linked in.

Each CUID is entitled to its own filesystem hierarchy which can be authenticated cryptographically. Normally, an MDDB peer will be created with the same CUID, to serve the filesystem, but an MDDB may host filesystems of more than one CUIDs. This helps when users cannot create and administer their own MDDB peers. Of course, since each resource has a single authority, there cannot be two MDDB peer with different CUIDs serving the same filesystem.

4.4.4 Storage Pool (SP) and Virtual Block Store (VBS)

Storage Pools implement the storage resource. A Storage Pool peer has access to raw block storage which exports via the Virtual Block Store (VBS) protocol. The storage unit will be called ***storage block***, but unlike the traditional usage of the term, block size may be dynamic or static at any value. In this sense, a storage block resembles a regular file, only it has another (limited) set of metadata. The variable block size does not introduce any problems because the DFS is a high-level systems that benefits from the low level storage management already established systems offer. These lower level systems take care of physical blocks (systems like disk filesystems or peer-to-peer block stores).

Storage Pools incorporate a Virtual Block Store layer (VBS) that handles storage allocation requests from peers. A VBS protocol is used for that purpose. The VBS protocol enables Storage Pools to logically aggregate several other Storage Pools' resources under their disposal. MDDBs are always associated with a Storage Pool were they can seek storage space when the MDDB clients don't provide their own.

4.4.5 Distributed File Service (DFS) Client

A DFS client is a peer that can access file and storage resources in a DFS network. DFS clients are controlled by users, directly via a local filesystem interface or indirectly, from inside a DFS-enabled application. Clients complete complex tasks like path walking, retrieval, caching and updating remote resources through a posix like interface, either integrated with the Operating System or in the form of a library. Clients also handle disconnected operation according to the users' settings.

4.4.6 Local FS Exporter

An important facility that promotes the value and usability of the architecture is exporting locally available filesystems and not having to copy them or create new ones. For this to be possible, the MDDB and the VBS must serve file metadata and data directly from the local filesystem, mapping local structures to DFS structures. Therefore a Local FS Exporter component includes an MDDB, a VBS and a local file access module, which all cooperate with each other.

4.5 Basic Functionality, Services and Features

This section describes the basic functionality and services provided by the DFS framework.

4.5.1 Overview

A DFS network consists of a number of DFS capable peers over the internet. There are three basic types of peers: MDDB's, Storage Pools, and DFS Clients. Each MDDB maintains its own filesystem namespaces and file metadata. MDDB contains pointers to the actual data storage. Each Storage Pool can be contacted for file data retrieval. DFS Clients access MDDB's and Storage Pools in the network in order to expose DFS resources to their local applications.

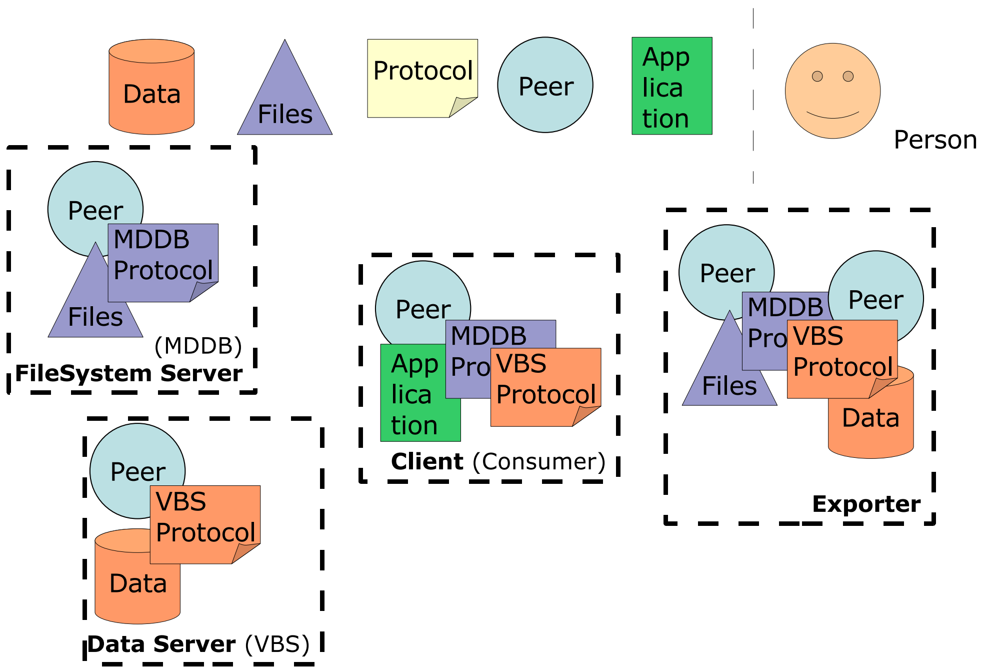

Figure 3: Participant components in a DFS network

4.5.2 Setting up an MDDB

A file system exists within the DFS network because an MDDB hosts it. Hosting a filesystem involves hosting the filesystem structure and file metadata, as well as serving filesystem access requests from other peers. Each filesystem has to be uniquely identified by an Owner, which essentially means that a new cryptographic certificate has to be created. This certificate grants unlimited access to the filesystem through the MDDB protocol.

To summarise, a filesystems comes into existence when someone:

- Spawns an MDDB daemon (or contacts a running one)
- Submits the filesystem Owner's certificate to the MDDB, or derivative delegation token.

4.5.3 Finding storage for the filesystem data

Setting up an MDDB may bring a filesystem to life, but the filesystem will be empty. In order to populate the filesystems, files must be created. Recall that the MDDB does not store file data but pointers to it[1]. In addition to file creation, data storage has to be allocated too. This is done via the VBS layer.

The VBS layer maintains a list of Storage Pools that have agreed to provide storage to it.

[1]Actually file data can be stored as "fake" metadata within the MDDB and then referenced via a special Storage Protocol

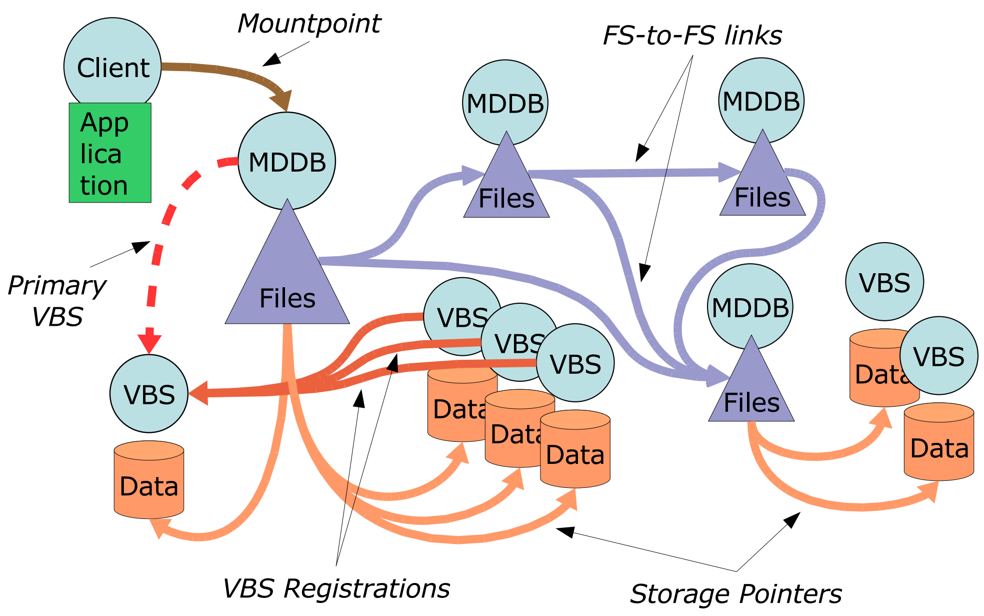

Figure 4: Schematic representation of a DFS network

4.5.4 Linking to other filesystems

Instead of pointers to file data the MDDB may contain a pointer to another file in the same, or different filesystem in the DFS network. This feature can be used to link from one filesystem to an other, much like the World Wide Web is interlinked. The most anticipated use of this linking is the aggregation of files from different peers under a single filesystem location.

4.5.5 Mounting DFS into local filesystem hierarchy

Mounting a remote filesystem resource means making that resource available within the local filesystem hierarchy. This involves contacting the remote source and retrieve the filesystem information and then present it to the local system via the native filesystem interface. The software components that are needed for this task are:

- A DFS Client for accessing remote DFS filesystem resources
- A native FS interface for presenting these resources to the native system.

When a DFS filesystem resource is mounted, the resource's URI is stored and all subsequent requests for files under the root of the mountpoint are prefixed with this URI. Each file is represented by a data structured specified in the MDDB protocol, called ***inode***. Additional specified data structure may complete the representation of a filesystem and its data. The DFS Client contacts the remote MDDB peers and acquires copies of these structures. These copies are then handled to the local FS interface for presentation to the applications.

4.5.6 Path walking

Once the DFS Client receives a root URI for mounting, it needs to contact the responsible MDDB for that URI. The filesystem URI contains the identity of the Owner of the filesystem. This identity is resolved to a network contact point through a global (or not global) index of users. This index may be a distributed hash table, that maps DFS CUIDs to network contact points. A ubiquitous file system operation will be this:

– Given a filesystem URI, retrieve the corresponding inode object.

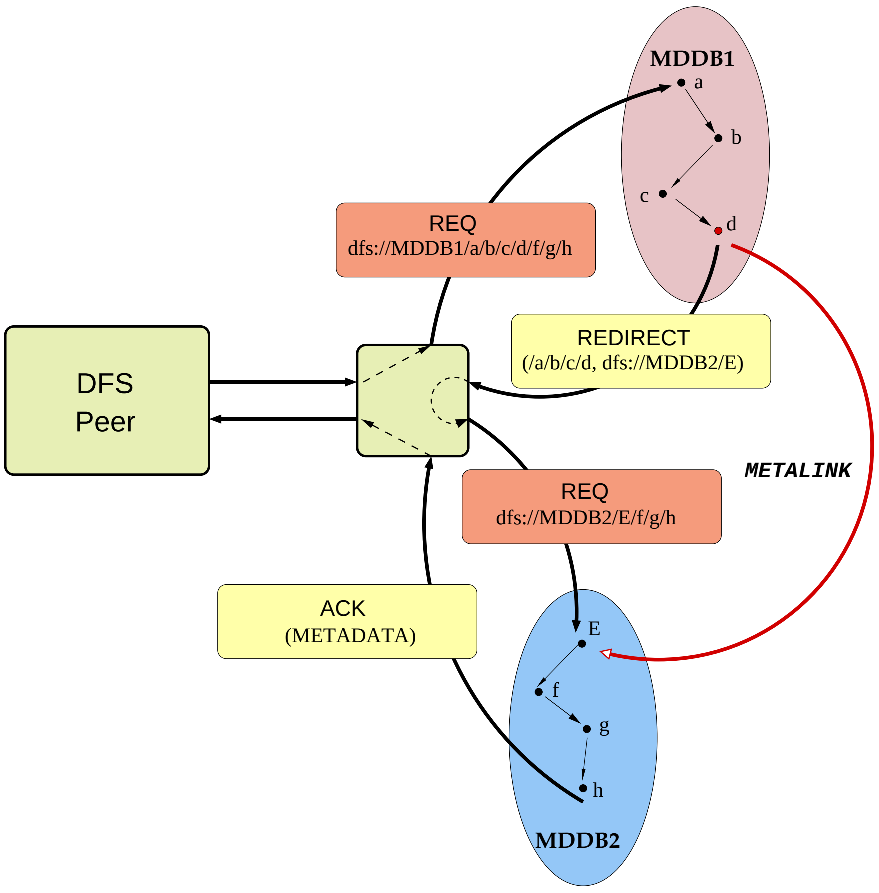

Figure 5: MDDB Path Walk

The inode object resides within some MDDB but because each filesystem can arbitrarily link to others, this MDDB is not known. What is known is the MDDB that maintains the filesystem of the Owner that is mentioned in the URI. With this MDDB as a starting point, the DFS Client asks for the inode corresponding to the whole path. The MDDB replies with an inode if it exists within its database, or with one path and one URI. The first path is the deepest path found within the database, while the URI specifies a link to another filesystem where the search for the inode should be continued. The DFS Client appends the unresolved path component of the original URI to the link URI and then, recursively, contacts the next MDDB. Eventually, the complete path has been resolved and the inode data can be retrieved. It should be noted that successive accesses to different MDDBs are completely independent as two different MDDB accesses.

4.5.7 Data Retrieval

After the inode data of a file is retrieved and are locally available, the file data can be retrieved using the storage pointers contained within the inode data. This involves successively resolving the storage URIs and contacting the corresponding storage Providers. From the retrieved data the contents of the file are reconstructed and saved locally.

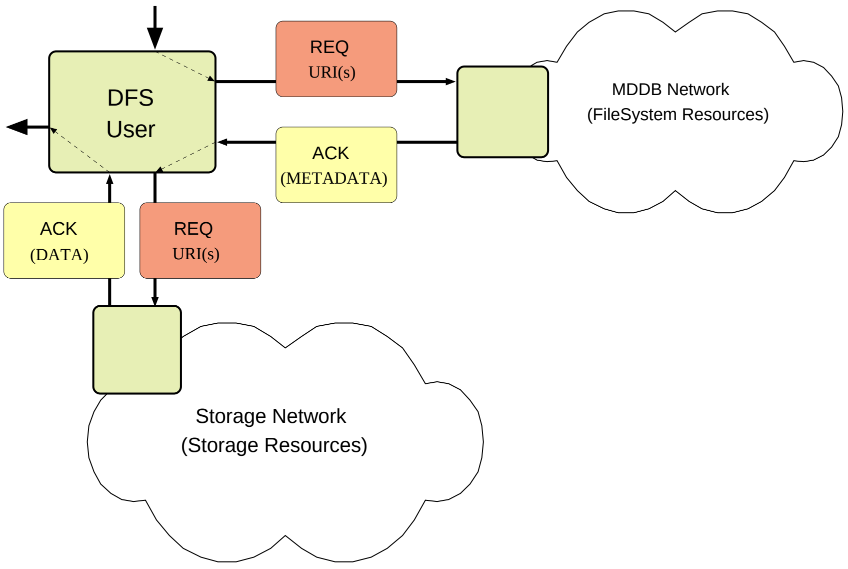

Figure 6: Data Retrieval

4.5.8 Data Modification

Because data storage is a separate resource from the filesystem, data can generally be read, written or modified regardless of any file association. File data are visible through the storage pointers in the inode data. Thus one can store somewhere one's data and subsequently modify the inode to update the storage pointers. But the system can be configured to accept only authorised storage providers and/or via a specific allocation process, possibly involving the local VBS layer of the MDDB. There are pros and cons for each approach:

i. DFS Client handles storage without asking for the MDDB's VBS's intervention

   + Huge amounts of data can be copied or created in arbitrary filesystems without any data transfers.
   - The DFS Client has more management overhead and has to have access to storage Providers.

ii. DFS Client requests storage allocation from the MDDB's VBS.

   + Clients are not burdened with the storage allocation overhead
   - Data have to be transferred

4.5.9 Local presentation of DFS

Local presentation of a DFS filesystem resource is really about integration with the OS. This typically involves implementing a virtual filesystem volume that can be *mounted* into the filesystem hierarchy. The OS file requests are translated into DFS Protocol requests and the results are properly reported back to the OS. The DFS Network is a peer-to-peer network because peers can arbitrarily link among them. There aren't any special resources – any URI can be mounted. The local filesystem interface, all it needs to do is store this URI to “pin down” it's view of the DFS Network using the URI as its root.

4.5.10 Exporting local filesystems to the DFS network

There are two basic approaches:

i. Local filesystem is copied to the local MDDB server and the two filesystems are strictly or loosely synchronised according to the needs. While the MDDB server is running, the MDDB version of the filesystem can be mounted over the original local filesystem. The local directory is hidden from the OS until MDDB is unmounted, when appropriate synchronisation happens.

ii. A special server, an ***Exporter*** exposes the local hierarchy in the network via the MDDB protocol but it directly uses the local filesystem for metadata and data storage. The MDDB version of the filesystem should also be mounted because the exporter will need to intercept all local accesses as well.

4.5.11 Pooling VBS resources

The VBS protocol allows VBS peers to accept registrations from remote storage providers. Each registration specifies the storage capacity provided and various storage attributes (e.g. quality) as well as the available access methods for it. VBS peers then aggregate resources registered to them by serving incoming storage allocation requests using any of the remote storage providers, in a way specified by policy (e.g. in a round robin fashion).

Remote providers may be VBS peers or other non DFS-aware types of storage services, such as FTP. VBS-to-VBS registration allows hierarchical management of storage resrources distributed accross a very large number of providers.

4.5.12 Allocation of storage in DFS

Allocation of storage is the responsibility of the file creator, since he controls the storage pointers within the file metadata. Nevertheless, the filesystem owner may want to arrange for specific allocation policy or quality, or just facilitate storage allocation on behalf of the users. For this reason, a VBS peer, the ***Primary VBS***, is always associated with any MDDB peer that serves a filesystem. Clients that cannot allocate their own storage can request storage from the primary VBS peer.

4.5.13 Contributing storage to specific users

Storage allocation is done via the VBS layer of the Storage Pool component. The VBS layer accepts registrations from other Storage Pools and maintains a contact list. Upon registering, a foreign Storage Pool submits information about the volume and the type of storage services they are willing to offer. The local VBS layer will allocate storage from its registered contacts to meet its own storage allocation needs.

If a user wants to contribute storage to another user, then it instructs his Storage Pool peer to register with its offering to the Storage Pool of the other user. Since each filesystem is always associated with a Storage Pool, for transparent file storage allocation, one can easily choose to contribute storage to a specific filesystem.

4.5.14 Contributing storage anonymously

Users wishing to openly contribute storage can register with a well known catalog of users that may be part of a VO resource management framework. Alternatively, the users can instruct their

VBS peers to participate in a global scale peer-to-peer block store network. See Section 6.3 for a relevant scenario.

4.5.15 Direct VBS access by applications

Applications that only need storage space for private use may allocate and access data directly from VBS peers, without having to use the allocated storage for any file An application may even share these storage pointers in its own arbitrary way.

4.6 Publish-Subscribe services

DFS Peers can subscribe to any object addressable by a URI and asynchronously be notified when events associated with the object occur. Event notifications can be reliably propagated and their delivery is deferred by the online status of a Peer.

There are two kinds of subscriptions to an object. The one delivers a notice about an event that occurred, creating a local event, or an activity log of the remote object. The other kind forwards the very actions received by the remote object, enabling a kind of selective state replication by locally executing them on an associated object.

4.6.1 Notification Subscriptions

A DFS Peer can subscribe to a remote object and receive notifications for actions that are received by the remote object or events that it undergoes. These notifications can be consumed to create arbitrary local events or be locally associated with an object that will log the activity.

4.6.2 Action Subscriptions

Action subscriptions are like notification subscriptions, but instead of delivering a short notification, the complete action that was received by the remote object is forwarded. The actions can be locally associated with an object that will selectively mirror the remote object's state. If the action semantics can be different for the two objects so that their state is not replicated but paralleled.

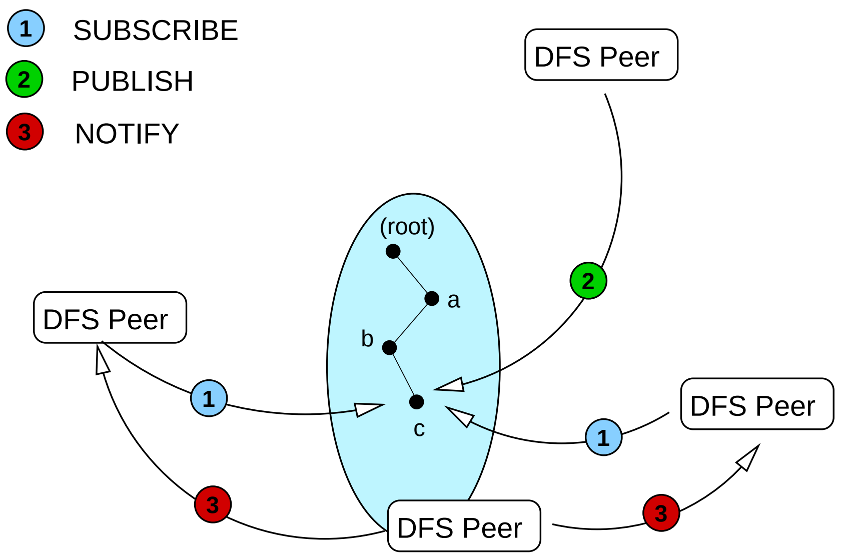

Figure 7: Notification mechanism

4.7 Replication

Replication in the DFS architecture may be found in several forms.

i. ***DFS Caching Replication*** refers to the local caching of remote files and objects that is performed by the DFS Clients in order to make the files faster to access and more available, even when disconnected. A Client that opens a file may subscribe for notifications so that its local copy eventually becomes synchronised with the remote file. The cached objects are not separate DFS resources and are not addressable from the DFS Network. The replicas of this type will be referred to as ***Local replicas***.

ii. ***DFS File Replication*** refers to the replication of a DFS file to other DFS files via the Action Subscription mechanism described earlier. The replicas are separate, addressable DFS files that can be independently administered. These DFS files will also be replicated in the Clients' local cache when opened. The replicas of this type will be referred to as ***Owned replicas***

iii. ***Anonymous Replication*** refers to the replication of resources that is performed in most DHT-like peer-to-peer networks. In this scheme, all replicas represent the same resource and are collectively addressable. DFS does not include such type of replication but underlying storage layers for VBS may replicate file data in this way.

   For example, VBS can operate on top of a peer-to-peer block store system that addresses a replicated block with its contents' hash. In this case, file contents are replicated in the storage layer resulting in increased file availability. Note that file metadata are not replicated in this way, but clients can retrieve the block lists from their cached local copies of remote files.

The following table summarises the features of the discussed forms of replication.

Replication form	Addressable	Separate Resource
Caching	No	No
DFS File	Yes	Yes
Anonymous	Yes	No

Table 2: Different forms of replication in the DFS architecture

4.8 Concurrency

Concurrent operations are serialised by the resource authority. Semantic conflicts are not handled by DFS, peers resolve these externally, maybe with the help of metadata stored in DFS. Locks and other exclusion mechanisms may be present to facilitate safety.

4.9 Disconnected operation

The DFS protocol mechanisms do not make any assumptions about the remote peer availability. Locally cached and modified objects, as well as communication messages persist until their synchronisation or delivery is acknowledged. The operations to the local state are unaffected by the disconnected status of the client.

4.9.1 Online files

Online files are files that use the publish-subscribe mechanism to selectively log or replicate remote file activity, or conversely, be logged or replicated.

5 The DFS+VBS Prototype

We have developed a DFS+VBS prototype that includes proof of concept implementations of the following features according to the DFS architecture previously presented:

– *MDDB, VBS, and Client peers*

– *POSIX interface integrated (mounted) under Linux OS*

– *Regular files, metalinks, virtual files*

– *Publish-subscribe services*

– *Update notifications on open files*

– *Disconnected operation (including update propagation) based on persistent caching*

5.1 Overview

Most of the design has been realised as software components that closely resemble distinct mechanisms and functionalities described in the architectural overview. The core of our implementation framework is the *DFS Peer* which provides the generic protocol handling functions in order to implement network communication endpoints. The DFS Peer manages and deploys software modules that implement services, depending on the role of the respective user in the DFS network. Such roles include:

- Filesystem resource ownership and provision, handled by the *MetaData DataBase (MDDB)* component. DFS users use their MDDB to host and export namespace structures to the overlay. Corresponding file name and metadata entries - including ACLs, pointers to storage facilities, etc. - may be physically stored at the host filesystem or a local SQL database server.

- Storage resource ownership and provision, handled by the *Virtual Block Store (VBS)* component. A VBS peer manages and exports locally attached raw block storage in behalf of a DFS user. It also handles remote allocation requests and logically aggregates all the local and/or distributed storage resources under the respective user's disposal.

- Resource consumption, through a *DFS Client Library* that provides external software the ability to access the DFS network of resources.

Peer inter-communication is achieved via generic *DFS Messages* that comply to either the *MDDB* or the *VBS* protocol. The DFS Message structure suggests a generic layout for protocols, based on request/reply verbs and corresponding arguments. MDDB and VBS protocols differ in the semantics of associated verbs and the handling mechanisms implemented in each component. The common message format permits peers to ignore protocols or protocol extensions they do not support or even to partially or abstractly parse messages.

DFS users materialise in the network as their corresponding roles are handled by service components deployed in DFS Peers. Services, named by URIs which include users' CUID identification and protocol description directives, can be located via the global DHT index.

Services in the DFS network are named by corresponding URIs, which include DFS User and service identification directives and can be located via the global DHT index. Each physical machine may host numerous DFS Peers in the same sense that each physical user may possess multiple DFS User identifiers (certificates). Some users may only provide storage, thus appear as a VBS-enabled peer in the network, while others may export files, storage and access the network — all through a single DFS Peer hosting all necessary components. Note that a user not owning resources may only use the Client Library, although deploying an MDDB and VBS will result in imposing authority to corresponding namespace entries and flexibility in the management of remote storage offerings.

The whole software architecture of the DFS prototype is based on the implementation of the DFS architectural objects. These objects can be handled within any software module and can be easily encoded to and from a serialised from.

Figure 8 visualises the software components of the prototype.

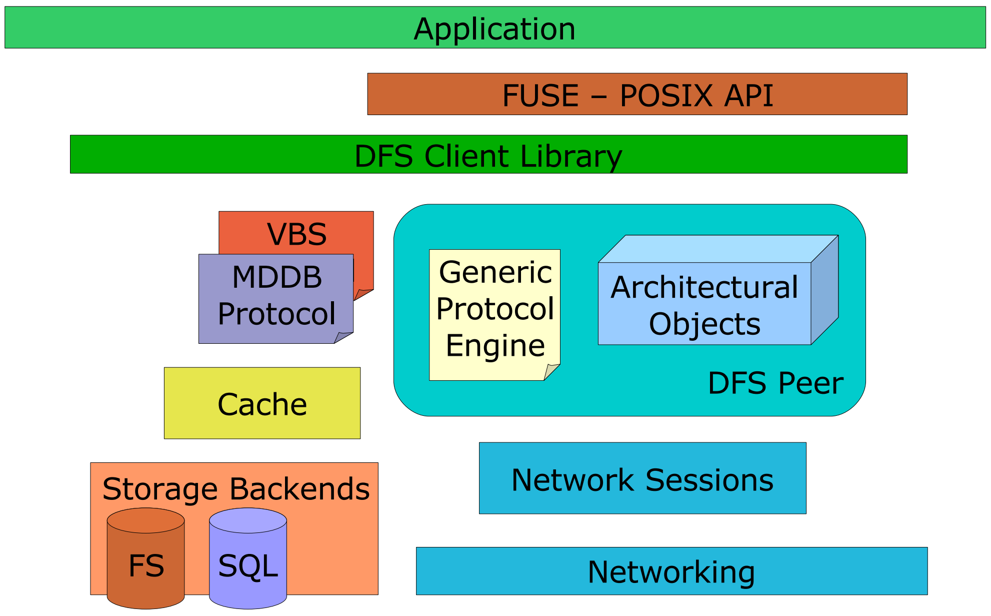

Figure 8: Anatomy of the DFS+VBS prototype implementation

5.1.1 The DFS Peer

The DFS peer operates in a context of its own. It communicates with any other software module through message queues. Message queues provide easy and dynamic interfacing with other software modules, and also facilitate serialisation of requests and isolation. The peer's message queue is an important part of the mechanism because it is the boundary of the context of the authority behind the peer.

The peer module implements a generic protocol engine. This engine can schedule requests for remote delivery and match their replies. This involves resolving the destination URIs, contacting remote peers and establishing sessions with them. The peer also maintains a persistent cache for pending requests. The protocol engine also accepts protocol handlers that implement peer to peer services like MDDB and VBS.

5.1.2 Network Sessions

The DFS peer communicates with other peers at session layer. These sessions are provided by a separate module that establishes sessions with remote peers over TCP/IP networking. A session can send and receive data asynchronously and it also provides notification of events and errors happening beneath, such as network disconnection or session loss. Sessions may also hold privately associated data.

5.1.3 Storage back ends

The DFS prototype needs to allocate and temporarily or persistently store various objects, such as file inodes (metadata), file data and protocol requests. There are several storage back ends that handle this. Back ends can be divided into two categories, those for storing inodes and the general purpose ones. All architectural objects can be stored in the general purpose back ends but inodes are better handled separately because of their hierarchical organisation and their direct correspondence to files.

Filesystem back ends Most of the storage back ends use a POSIX filesystem to store objects. General purpose ones hash the object's identifier to obtain a path and then they store the objects contents in the file's contents. Inode storage back ends use the inode's URI converted to a path.

Special filesystem exporter back end A special filesystem exporter inode back end was developed to enable the exporting of existing locally available filesystems into the DFS network. In this scheme, file hierarchy and some other relevant attributes are encoded in the underlying POSIX filesystem's metadata, while the remaining DFS-specific attributes are stored like with the normal inode stores. The normal inode store is kept in another filesystem hierarchy so that the exported filesystem remains untouched by the DFS prototype.

SQL back end For storing the MDDB metadata, an SQL back end was developed, while not as versatile as the filesystem back ends, it offers superior performance and complexity in querying the hierarchy and other metadata.

5.1.4 Cache

On top of the storage back ends, a caching layer was implemented so that remote objects can be locally available. The cache has a persistent option that is used for disconnected operations. There are also lru options for replacing objects in the cache. The cache normally keeps volatile (non-persistent) objects in memory with the exception of file data which are stored in disk due to their size. Fast access is still available for on-disk objects because the operating system caches disk accesses independently.

These caches are not intended to be consistent with remote states. Their purpose is to make documents available for speed and disconnected operations. However, with proper subscriptions to remote files, these caches will be updated from the remote files in a best effort manner.

5.1.5 Operating system integration

The DFS Peer's Client Library exposes a high-level API that presents the peer-to-peer namespace and storage through traditional POSIX-like calls. Additionally, we have implemented a FUSE filesystem [43] that uses this API and allows "mounting" an MDDB hierarchy alongside local files in a Linux system. Normal applications can then transparently access the DFS Network via a file interface regardless of whether those files and their corresponding data is local or remote.

When viewing DFS through FUSE, metalinks appear as regular symbolic links, only with special naming semantics. The mounted DFS namespace, rooted at the local filesystem path specified at mounting time, includes a specific MDDB's hierarchy. Nevertheless, it also provides access to any other URI in the DFS Network through special paths. A virtual directory, named '`@`', is present in every FUSE directory, enabling the creation of metalinks via symlinks with the idiom `@/URI` as their target. Thus clients can navigate freely throughout the DFS network. Request for `@/URI`-like paths anywhere in the FUSE filesystem will dereference the URI and access it directly over the network. Likewise, local filesystem symbolic links can refer to any DFS file by prefixing the URI with the FUSE mountpoint and the `@` directory.

The `@` directory is actually a special case of a *virtual file*. Virtual files are special files that are not part of DFS file data or metadata; their contents are computed on demand. They are used to access file metadata and unique operations of the DFS implementation through the traditional filesystem API (provided by the FUSE layer). Virtual files are formatted as: `<base path>@<virtual path>`. The `<base path>` signifies the actual (non-virtual) path that the virtual file is associated with. The `<virtual path>` can either be a name (if the virtual file is a plain file), or a whole hierarchy (in the case the virtual file is a directory). For example, information about the DFS-specific status of a file is accessible via `file@status`. Writable virtual files can be used to input data such as configuration options or DFS-specific commands. For example, the `file@config` file can be used to manipulate many aspects of a file's status. Virtual files in the current prototype implementation are used mainly for managing caching and disconnected operation and providing interface to special communnications and file monitoring functionality. The visibility of virtual files is also configurable.

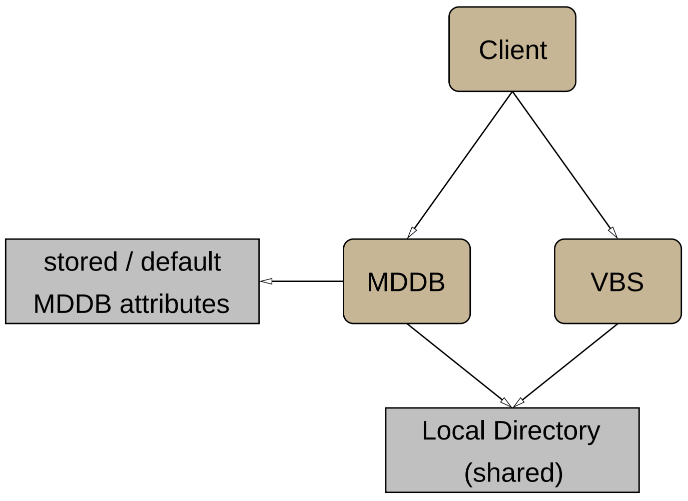

Figure 9: Exporting existing filesystems

5.2 POSIX interface

The application interface to the DFS prototype is a POSIX-like filesystem, mounted locally under the Linux operating system. The filesystem is enhanced with *virtual files*, that provide access to information and functionality specific to the DFS. Virtual files are artifacts of the interface and do not represent DFS resources.

The mounted DFS namespace will appear as a normal filesystem. All common file operations will be transparently translated to DFS operations. The tables 5.2 and 5.2 list some of the common POSIX calls that applications may use. Note that the lists are not final.

⋆ `open()`
The filesystem creates a handle for the open file. This handle is associated with networking connections, cached file data and other DFS objects (e.g. credentials)

⋆ `close()`
The filesystem destroys the handle for the open file after writing back any modified cached data and objects.

⋆ `read()`
Data are read from the network and returned to the user. Data may be cached or prefetched for performance reasons.

⋆ `write()`
Data are written to the file. The remote file may not be updated until a `close()` or `flush()` call.

⋆ `lseek()`
The file pointer is moved to the specified offset.

⋆ `opendir()`
The filesystem creates a handle for the open directory. This handle is associated with networking connections, cached file data and other DFS objects (e.g. credentials)

⋆ `closedir()`
The filesystem destroys the handle for the open directory after writing back any modified cached data and objects.

⋆ `readdir()`
Reads the contents of the opened directory. Data may be cached or prefetched for performance reasons.

⋆ `flush()`
Causes locally modified data to be sent to the network.

Table 3: POSIX calls for file access.

⋆ `mkdir()`
The filesystem creates a new directory.

⋆ `rmdir()`
Removes a directory from the filesystem.

⋆ `symlink()`
Creates a Metalink DFS Object.

⋆ `creat()`
Creates a new file.

⋆ `unlink()`
Removes a file or directory from the filesystem.

⋆ `stat()`
Get a file status. Data may be cached or prefetched for performance reasons.

⋆ `statvfs()`
Returns file system information.

Table 4: POSIX calls for filesystem manipulation.

5.3 VBS Lower Programming Interface for storage drivers

The VBS module can accept arbitrary storage layer beneath it. The current implementations include a disk storage driver and a peer-to-peer block store driver. Storage drivers are run within the context of the DFS Peer module and thus can access all peer facilities for sending and receiving messages to other peers. Therefore, to implement a storage driver for VBS it suffices to implement handlers for the VBS protocol requests.
The VBS protocol is outlined here:

⋆ `RETR <block> [<offset>] [<size>]`
`Retrieves a storage block.`

Parameter	Description
`<block>`	Identifies the storage block to be retrieved. Mandatory.
`<offset>`	Specifies the offset of the data to be retrieved. Optional. If the block is seekable the default value is the current seek position else the default is zero.
`<size>`	Specifies the size of the data to be retrieved. Optional. The default value is the remaining size of data beyond the offset.

⋆ `STORE <block> <data> [<offset>]`
`Stores a storage block.`

Parameter	Description
`<block>`	Identifies the storage block to be stored. Mandatory.
`<data>`	A container encapsulating the data to be stored inside the storage block. Mandatory.
`<offset>`	Specifies the offset inside the storage block for the data to be stored. Optional. If the block is seekable the default value is the current seek position else the default is zero.

⋆ `ALLOC <size> [<options>]`
`Allocates space for a storage block.`

Parameter	Description
`<size>`	Specifies the size of the storage block that is requested to be allocated. Mandatory.
`<options>`	Options passed to the VBS Server to define special allocation properties. Optional.

⋆ `DEL <block>`
`Releases space previously allocated for a storage block.`

Parameter	Description
`<block>`	Specifies the storage block to be deallocated. Mandatory.

⋆ `REGISTER <contact> <size> [<options>]`
`Registers a VBS Server as a Provider for another VBS Server.`

Parameter	Description
`<contact>`	The contact of the VBS Server acting as Provider. Mandatory
`<size>`	The size of space provided by the VBS Server. Mandatory
`<options>`	Options to define special registration properties. Optional.

⋆ `UNREGISTER <contact> [<options>]`
`Unregisters a VBS Server previously acting as a Provider.`

Parameter	Description
`<contact>`	The contact of the VBS Server willing to unregister. Mandatory
`<options>`	Options to define special registration properties. Optional.

⋆ `ACK [<return value>]`
`Acknowledges the successful execution of a command.`

Parameter	Description
`<return value>`	A container encapsulating the value returned from a previous command. In the case of a RETR it contains the data of the requested block. In the case of alloc it contains the handle of the allocated storage block. Optional.

- ⋆ `ERR [<description>]`
  `Signifies erroneous execution of a previous command.`

Parameter	Description
`<description>`	A description of the error. Optional.

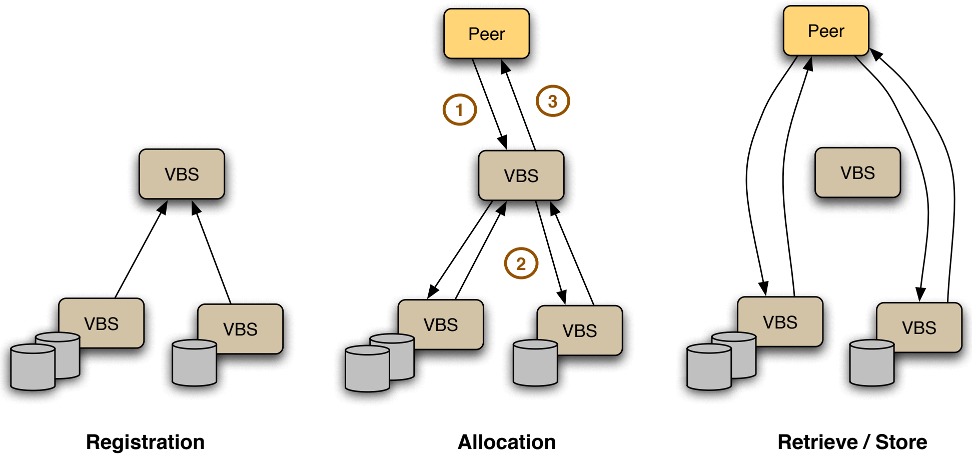

Figure 10: VBS registration and storage allocation

5.4 Functionality

DFS Peers include a cache component to temporarily store frequently-accessed remote objects (file metadata and contents). Users may also explicitly request that certain files remain cached through the client API or a virtual file (when using the FUSE filesystem). By exploiting the cache module we also have an initial implementation of a disconnected operations component. To support disconnected operations, DFS protocol mechanisms do not make any assumptions regarding remote peer availability. Locally cached and modified objects, as well as communication messages remain in pending state until their synchronisation or delivery is acknowledged. Also, a persistent storage module allows such objects to survive peer downtime.

DFS Peers can also subscribe to any object addressable by a URI and be notified asynchronously when events associated with the object occur. Event notifications can be reliably propagated and their delivery is deferred by the online status of a peer. There are two kinds of subscriptions to an object. One delivers a notice about an event that occurred, creating a local event, or *activity log* of the remote object. The other forwards the very actions received by the remote object, enabling a kind of selective state replication by locally executing the same actions on an associated object. File replicas, whose consistency is maintained through the action notification mechanism, are called *online files*.

5.4.1 Virtual files

Virtual files are special files that exist only as part of the DFS POSIX interface and provide access to DFS-specific functionality.

The `@/` directory	This special directory is accessible within any non-virtual directory. It appears empty but all paths below it are interpreted as URIs that provide direct access to remote destinations.
`@ state`	Displays the current client-side state of file. There are several aspects of this state such as *disconnected, persistent* states.
`@ config`	Receives configuration commands for the client-side state that is reported via the `state` file.
`@ info`	Reports various low level attributes about resources, including storage locations where the file data are stored. This differs from the `status` virtual file in that it gives information about the corresponding DFS resource and not about the client-side state of accessing it.
`@ attr`	Receives configuration commands for low level attributes that are reported via the `attr` file.
`@ disconnected`	This file is not for reading or writing. It is used to alert the user that the current directory is disconnected. If a directory becomes disconnected (after a timeout), then this file appears. Removing it instructs the system to try and reconnect. Normally, attempts to reconnect will be scheduled at exponentially growing intervals.
`@ publish`	Provides a communications channel among the subscribers of a file for broadcasting private messages.
`@ notify/`	This special directory includes a control file, `ctl` that is used to create and configure new virtual files for receiving notifications in a specialised way

Table 5: Virtual files with their description.

5.4.2 Monitoring files

Monitoring files is achieved through a virtual file interface. Through a file's `@notify/ctl` virtual file, an application may setup to receive a signal every time an event has occured. Applications supply a list of events to be selected for notification.

5.4.3 Disconnected operation

Clients access remote resources through their local cache. This means that files can remain accessible when disconnected as long as they are cached. Caches use an LRU policy for content replacement but not all files are candidates for eviction from cache. Caches will not evict critical files, such as modified files that have not been written back or files that are explicitly marked as persistent. Moreover, critical files will always be stored persistently so that they can survive a system restart and possibly a failure.

5.5 Authentication and Authorisation

While internal structures and mechanisms have been designed with authorisation in mind, no detailed design or support in the prototype exists. The main issue is that Authentication and Authorisation are tightly coupled with VO management. Therefore, design and implementation of authentication and authorisation were planned to start after VO management has produced relevant specifications.

5.6 A peer-to-peer block store implementation of VBS

To experiment with the scenario of implementing VBS with a peer-to-peer block store (see Section 6.3), a Kademlia [44] distributed hash table implementation was adapted and used as a special storage subsystem that can be used by a VBS peer in place of its local storage. The VBS peer spawns a Kademlia peer that in turn joins the Kademlia network. The VBS peer translates access to storage pointers in the VBS protocol, into access to content-hash blocks in the Kademlia DHT. VBS peers of this type contribute storage and provide access to the shared distributed storage pool created by the Kademlia DHT.

6 Usage Scenarios

6.1 A VO scenario – Creating policy over DFS

This section proposes a solution for VO Filesystem management over the Distributed File Services architecture.

There are several kinds of users involved:

- ***VO users*** are the members of a Virtual Organisation. They are identified via cryptographic certificates.
- ***DFS users*** are those who operate DFS Peers. They are cryptographically identified by their CUIDs.
- ***Local users*** are users of the Operating Systems or applications. They are identified with application-specific credentials.

Users in any of these systems can be created and destroyed. Local users are completely in local control while VO and DFS are controlled by whoever owns their identifying private/public key pair. The following requirements are posed by the nature of large peer-to-peer networks:

1. Local users can and do completely control their local resources and policies – *Independent administrative domains*
2. Local users need to participate in larger groups pursuing a common goal – *A need to form Virtual Organisations*
3. Groups of users have their own policies and resources, created from peer resources upon agreement – *A need to aggregate and manage VO resources*
4. In order to be true peers, local users must be able to control their resources and the authenticity of their content inside and outside the user groups they are members – *A need to control their own identity in the network*

GRID solutions historically tend to sacrifice peer freedom in order to enforce policy and safeguard collective goals. Peer-to-peer solutions, on the other hand, traditionally emphasise peer freedom. The DFS architecture proposes an effective combination of the two, that seems intuitively appropriate from a Grid4All perspective:

- Each peer has its own local authority over its filesystem resources.
- Each filesystem has the ability to directly link to other peer's filesystems

- When a group needs a central authority, for example a VO, a new peer is created and managed (outside of the DFS Network). This VO peer is now the authority of the group's filesystem resource. Local administrative control is preserved because peer contents remain local while central authority is established because the authoritative VO peer, links into is member's resources to form the aggregate VO filesystem.

As a more concrete example consider this scenario, that is graphically presented in figure 11

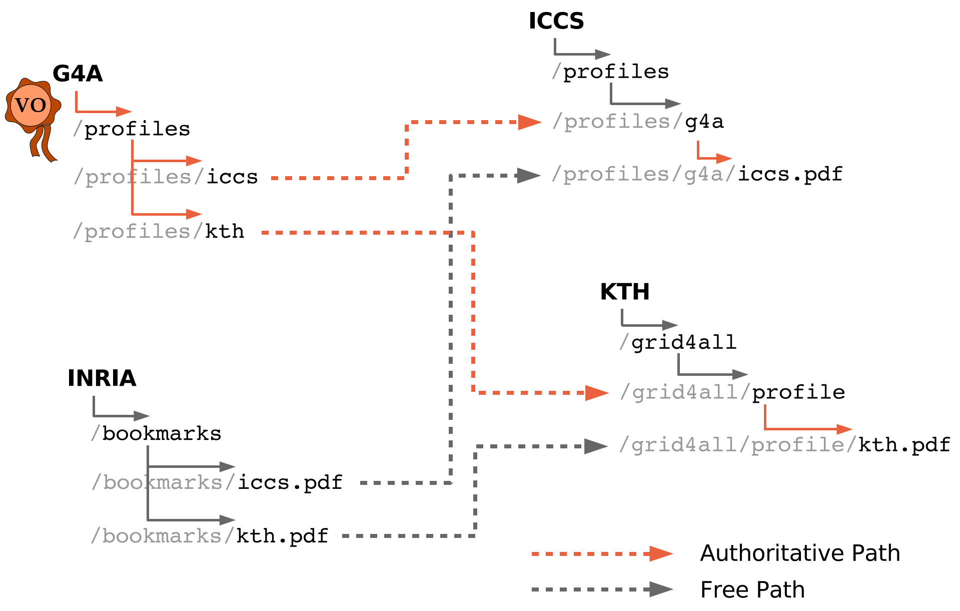

Figure 11: Authoritative and Free Paths in the DFS Network

- KTH, ICCS and INRIA decide to form a VO named 'G4A'
- KTH and ICCS have their profile documents in their DFS filesystems,
  `ICCS:fs/profiles/g4a/iccs.pdf` and
  `KTH:fs/grid4all/profile/kth.pdf`
- As a VO resource, these documents (or rather, the directories that contain them) are listed under the G4A filesystem as
  `G4A:fs/profiles/iccs` and
  `G4a:fs/profiles/kth`
- ICCS and KTH maintain the authority over their profiles while, through the links from the authoritative peer G4a, their documents are officially recognised as the G4A ICCS and KTH profiles.
- A third party, INRIA can also link to the same files from its filesystems' bookmark directory. These links are fully functional but they lack the authenticity of the G4A. Only the G4A peer controls what is to be linked as official VO resource and only the Owner of this resource can author its contents.
- Content may be available through ad-hoc ***free paths*** in the DFS network, while official and authentic content can be reached through ***authoritative paths***

- Effectively, DFS provides flexible mechanisms for constructing a peer-to-peer distributed system with the capability for central authorities.

6.2 Multilogs over DFS

This section presents two similar approaches to the implementation of multilogs over DFS and describes how the requirements from the various usage scenarios can be satisfied. Both approaches use the same DFS mechanisms. The difference lies in the layer that implements the POSIX interface.

6.2.1 Concepts and terms

Event and Action Notifications. Via a publish-subscribe mechanism, a DFS Client can subscribe to receive notifications when an event happens on a remote object.

- ***Event Notifications*** signify that a particular event has happened on the remote object.
- ***Action Notifications*** deliver the specific action that was performed to the remote object. Actions are defined by the DFS Protocol in effect. The actions semantics are defined by the receiving DFS Object. The objects' state is changed by the actions that the various agents perform upon them.

Disconnected Operation. Disconnected operation refers to the ability of the DFS Client to operate normally when the network is unavailable. The disconnected operation essentially is an extreme case of latency tolerant operation, when the latency may be indefinite.

Online files. Online files are files that use the publish-subscribe mechanism to selectively log or replicate remote file activity, or conversely, be logged or replicated.

Replication. Replication may have many forms in this context. The following list differentiates among the various forms that will be discussed in this section.

- ***DFS Caching Replication*** refers to the local caching of remote files that is performed by the DFS Clients in order to make the files faster to access and more available, even when offline. A Client that opens a file may subscribe for notifications so that its local copy eventually becomes synchronised with the remote file. The cached files are not separate DFS resources and are not addressable from the DFS Network. The replicas of this type will be referred to as ***Local replicas***.
- ***DFS File Replication*** refers to the replication of a DFS file to other DFS files via the Action Notification mechanism. The replicas are separate, addressable DFS files that can be independently administered. These DFS files will also be replicated in the Clients' local cache when opened. The replicas of this type will be referred to as ***Own replicas***
- ***Anonymous Replication*** refers to the replication of resources that is performed in most DHT-like peer-to-peer networks. In this scheme, all replicas represent the same resource and are addressable. DFS does not include such type of replication.

The following table summarises the features of the presented forms of replication.

Replication form	Addressable	Separate Resource
Caching	No	No
DFS File	Yes	Yes
Anonymous	Yes	No

Table 6: Different forms of replication

Multilog. The ***multilog*** is a special type of file that represents a logical document concurrently edited by several users via the Semantic Store layer. From the viewpoint of the Semantic Store application, the multilog is a set of logs, one for every editor of the document. Each user writes to his own log and reads from all the others. The application may associate more files with the logical document, but these files are private to each user.

Communication Channel. The internal communications device that is used to broadcast updates to all the editors of a multilog document will be exposed to the interface so that applications can broadcast and receive private messages. This facility will be called the ***communication channel*** of the multilog.

6.2.2 Multilog replication

A multilog file is shared by several editors. Each editor must have copies of all the others' logs locally available. This can be implemented with both DFS replication forms discussed – the Caching Replication and the File Replication. In the first case, the multilog is a set of regular DFS files. In the second case, the logs are DFS online files. The following paragraphs list the main features of a multilog implementation according to each one of two available replication forms.

6.2.3 The multilog as a set of regular DFS files.

- it can be replicated to all its editors' local machines with the DFS file replication mechanism
- the application will be able to access the multilog's files even when offline and all updates to the editors' logs will be visible via automatic notification subscription upon file opening.
- Per-editor files can be located in editors' directories below the multilog root and as long as an editor can edit only his own files, there will be no consistency problem.
- The authoritative multilog version is rooted at the specific peer that hosts the multilog's URI. Using links, editor files can be hosted anywhere.
- All file operations are sent to the host of the file and then propagated to all peers that have opened it.
- A user can access the multilog files from anywhere as long as the peers that host it remain online. In this scenario, both replication forms are identical.

Figure 12 displays a scenario for implementing a multilog over regular DFS files using DFS virtual files for special purpose operations.

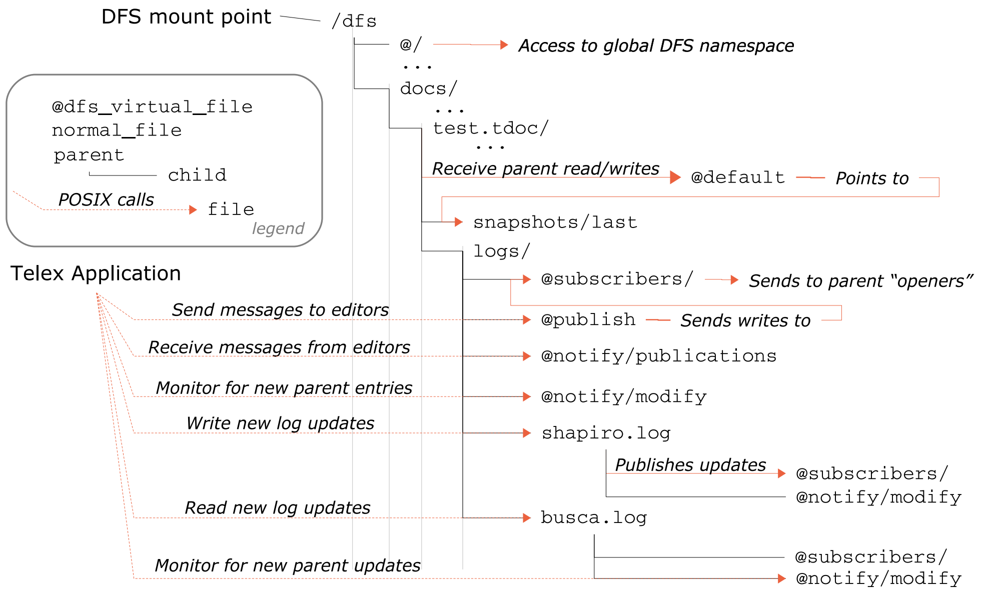

Figure 12: A possible implementation of multilogs on DFS

6.2.4 The multilog as a set of online DFS files.

This approach introduces another level of redirection and makes the multilog files replicated in the filesystem level:

- Each editor owns his own replica for each one of the other editors' logs.
- The shared namespace is used to list by linking to each editors' multilog hierarchy.
- The per-editor replicas are implemented with DFS online files. These files are connected via the publish/subscribe system to their authoritative sources within the other editors' multilog roots. This brings the replication to the DFS level. Local replicas become Own replicas and refer to different DFS resources. Own replicas can be made persistent by hosting them in a highly available peer.
- Local replicas of Own/persistent replicas are still created in the local machine of an editor when he accesses his multilog. He can edit his multilog offline. When back online, he will update his multilog and update his local replicas of its file. Because his multilog's file are online and connected with the other multilogs' files, updates will be propagated to all. The editors' multilog itself may become offline and then come back online to receive and send updates. In general, any part of the system can become offline without causing disruption.

  The added benefits of this approach are:
- Multilog replicas are addressable as separate DFS resources.
- Each user can completely control his multilog.

  In the first scenario, if a user deletes his log, then it becomes unavailable to all others. It could still be accessible via the local replicas but that would introduce a semantic problem, since the local replica refers to a resource that has been destroyed so it has to be destroyed, too. If we would like the replica to persist then we have to disengage it from the

DFS resource it refers to. That means that we have to create a separate resource, just like the second implementation does.

In the second scenario, a user controls the applicable modifications to his multilog. He may choose to accept all append operations but to allow none other. Even other users become hostile against him they cannot affect the view of the multilog he has already attained.

- Multilog replicas as DFS resources allow the user to use the DFS for the storage provision and allocation he needs for working.
- A generally more peer to peer approach for multilogs. For example, a peer can allow any editor into his multilog even if the multilog administration does not permit it, or even know about it. There is no physical centralised entity that implements the multilog.

6.2.5 Disconnected operation

DFS Clients locally cache remote resources that are accessed by the applications and use this mechanism to provide disconnected operation. The application can continue to work with the resources as long as they do not require an acknowledgement for the update of any remote resources.

6.2.6 Availability

Availability of files in the DFS Network is by design dependent on the availability of the serving Peer. A user requiring availability that his resources cannot provide must host his files in a highly available server that will be provided as part of the policy of the system.

6.2.7 Mobility

The mobility requirement is a combination of disconnected operation and availability. Disconnected operation is always available. If a user requires mobility, his files must be hosted by a highly available peer.

6.2.8 Moving files

[Moving files in DFS generally has the expected results but there are several subtleties. The sources of these subtleties are the identification of the files by their URI and the non-conventional, distributed nature of the DFS. Semantics have not been defined yet but almost any desirable behaviour can be emulated.]

6.2.9 Sharing multilogs

If multilogs are implemented as a set of regular DFS files, then sharing is straightforward. The normal mechanisms provide access to all files in the network. If multilogs are implemented with online files, then an index has to be maintained in order for the editors to be able to identify each other. This index can well be standard DFS directory with links to the editors' multilog replicas.

6.3 Peer-to-peer block store under VBS

In this scenario, a group of users wishes to aggregate their storage resources in a reliable and available pool. From that pool, each one will privately allocate and access storage according to his needs that may exceed those of any single user's capability. The users need this storage to be available and tolerant to individual network or storage failures.

Therefore, they form a Virtual Organisation that includes a peer to peer block store network. Then they instruct their VBS Peers to contribute to and request storage from the block store. Thereafter, requests to the VBS Peers for storage allocation and access Peers may be served from the peer-to-peer block store. The DFS system inherits the storage availability and scalability of the peer-to-peer block store while it maintains the ability to allocate storage in the normal way.

Furthermore, external users that are not members of the task force but wish to contribute storage for the VOs cause, can just join the block store network.

Another extension to this scenario is that of a global peer to peer block store that can support storage needs for a vast DFS network consisting of many independent groupd and individuals that all want to anonymously contribute to the global storage pool.

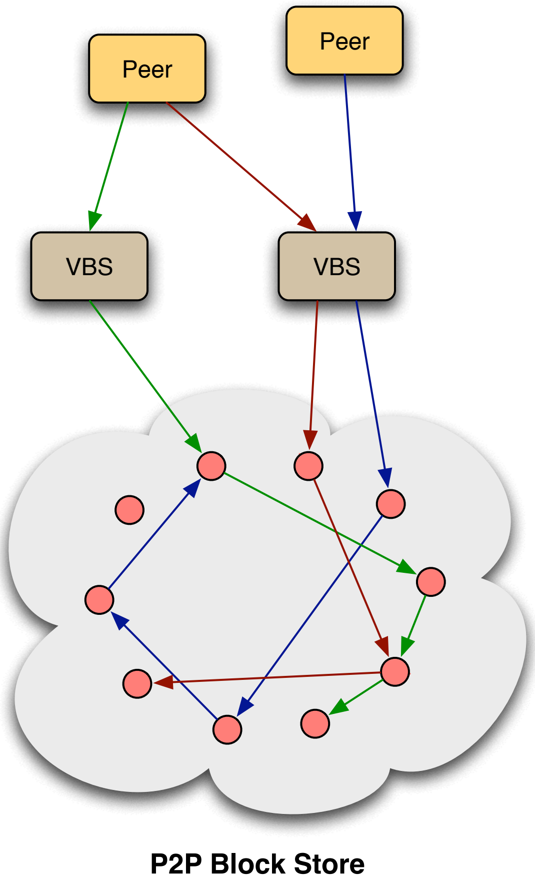

Figure 13: VBS over peer-to-peer block store

7 Relationship with other tasks

This section references dependencies and other issues regarding task 3.1 and other tasks.

7.1 T1.1: Overlay Services

The DFS architecture needs a naming service — a global CUID index — so that URIs are resolvable to actual authentication and network location data. Using a DHT is an appropriate solution in terms of resilience, availability and scalability of this index. Task 1.1 plans basic key-value lookup, membership and message delivery services as part of their overlay services infrastructure. These services may be used to implement the global CUID index, which would help integrate DFS better to the rest of the Grid4All middleware, as well as help exploit the capabilities of the developed overlay infrastructure.

7.2 T2.2: Market Based Resource Management

Task 2.2 is to create a market based resource management infrastructure. VBS peers are the storage providers for file data. VBS peers themselves aggregate storage resources from other VBS peers or external storage providers (see Section 4.5.11). In the context of Task 2.2, this scheme corresponds to a storage market where VBS peers can seek to buy storage from other VBS peers or external storage providers. Therefore, it is planned to use Task 2.2 marketplaces for trading storage, as this will add significant value to the DFS substrate.

7.3 T3.3: Semantic Store

Task 3.3 enhances regular file storage with semantic functionality that allows applications to define constraints within and accross documents and then manage and resolve conflicts that occur from shared use of these. Semantic Store will use DFS to implement a special type of file, the *multilog*, that will represent those documents. There has been extensive collaboration between T3.3 and T3.1 so that all required functionality is available through the POSIX interface of the DFS. Details of a multilog scenario can be found in Section 6.2.

7.4 WP4: Applications

Work Package 4 is about applications that will be developed to use the Grid4All environment. Applications require storage for files in a distributed environment, possibly with the option to automatically search and allocate storage resources. Work Package 3 plans to offer distributed storage of regular files through DFS combined with the more sophisticated document types of the Semantic Store. The POSIX interface makes the use of DFS from the applications, straightforward. Semantic Store files are also easily accessible because they are implemented within the POSIX interface of the DFS.

References

[01] Storage Networks: The Complete Reference. McGraw-Hill Osborne Media, 2003

[02] Network Attached Storage Architecture. Garth A. Gibson and Rodney Van Metter, Communications of the ACM/Vol 43, No 11, 2000

[03] RFC1094: NFS: Network File System Protocol Specification. Sun Microsystems, Inc. , March 1989 http://www.ietf.org/rfc/rfc1094.txt

[04] RFC1813: NFS Version 3 Protocol Specification Callaghan, el al, June 1995 http://www.ietf.org/rfc/rfc1813.txt

[05] RFC3530: Network File System (NFS) version 4 Protocol. Shepler, et al. , April 2003 http://www.ietf.org/rfc/rfc3530.txt

[06] RFC3720: Internet small computer systems interface (iSCSI). http://www.ietf.org/rfc/rfc3720.txt

[07] InfiniBand Architecture Specification. Volume 1, Release 1.1, Infiniband Trade Association

[08] Fibre Channel for Mass Storage. Ralph H. Thornburgh, Prentice Hall PTR, 1999

[09] ATA over Ethernet. Protocol Specification
http://www.coraid.com/documents/AoEr8.txt

[10] High Availability Data Replication. Paul Clements, James E.J. Bottomley Proceedings of the Linux Symposium, Ottawa, Canada, July, 2003

[11] Performance Evaluation of Parallel I/O in Cluster Enviroments. Sumanth J.V, Xiao Qin, Ala Qadi and Hong Jiang Technical Report TR-UNL-CSE 2002-0502, May 2002

[12] HyperSCSI Protocol Specifications. Data Storage Institute, Singapore `http://www.dsi.a-star.edu.sg/Library/Files/NST_Download/documentation/HyperSCSI_Protocol_Specific.pdf`

[13] A Cost-Effective, High-Bandwidth Storage Architecture. Gibson et al. Proceedings of the 8th international conference on Architectural support for programming languages and operating systems (ASPLOS), October 02 - 07, 1998.

[14] Petal: Distributed Virtual Disks. Edward K. Lee and Chandramohan A. Thekkath, roceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1996.

[15] The Part-Time Parliament. Leslie Lamport, Technical Report 49, Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301-1044, September 1989.

[16] Chained declustering: A new availability strategy for multiprocessor database machines. Technical Report CS TR 854, University of Wisconsin, Madison, June 1989.

[17] Semantically-Smart Disk Systems. M Sivathanu et al., 2nd USENIX Conference on File and Storage Technologies (FAST) 2003

[18] A Human Organization Analogy for Self-* Systems John D. Strunk, Gregory R. Ganger Appears in Proceedings of the First Workshop on Algorithms and Architectures for Self-Managing System, 11 June 2003

[19] Self-* Storage: Brick-based storage with automated administration. Gregory R. Ganger, John D. Strunk, Andrew J. Klosterman Carnegie Mellon University Technical Report CMU-CS-03-178, August, 2003.

[20] Building Distributed Enterprise Disk Arrays from Commodity Components. Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2004

[21] A Decentralized Algorithm for Erasure-Coded Virtual Disks. Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch, Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04), 2004

[22] Kybos: Self-Management for Distributed Brick-Based Storage. IBM Research Division, RJ10356 (A0508-013) August 26, 2005

[23] The Global File System: A File System for Shared Disk Storage. S. Soltis and G. Erickson and K. Preslan and M. O'Keefe and T. Ruwart submitted to the IEEE Transactions on Parallel and Distributed Systems, October 1997

[24] A 64 Bit, Shared Disk File System for Linux. Kenneth W. Perslan IEEE Mass Storage Systems Symposium, San Diego, California. 1999

[25] The Coda Distributed File System. Peter J. Braam, Linux Journal, http://coda.cs.cmu.edu/ljpaper/lj.html

[26] Lustre: A Scalable, High-Performance File System. Cluster File Systems, Inc. http://www.lustre.org/docs/whitepaper.pdf

[27] Lustre: Building a File System for 1,000-node Clusters. Philip Schwan, Proceedings of the Linux Symposium, Ottawa, Canada, July, 2003.

[28] Towards an Object Store. A. Azagury et al., 20th IEEE Symposium on Mass Storage Systems (MSST), 2003.

[29] zfs - A Scalable Distributed File System Using Object Disks. had Rodeh and Avi Teperman, 20th IEEE Conference on Mass Storage Systems and Technologies (MSST), 2003.

[30] Portals 3.0: Protocol Building Blocks for Low Overhead Communication. R Brightwell et al., Proceedings of the 16th International Parallel and Distributed Processing Symposium, (IPDPS), 2002.

[31] PVFS: A Parallel File System For Linux Clusters. P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, Proceedings of the 4th Annual Linux Showcase and Conference, 2000.

[32] PVFS2. http://www.pvfs.org/pvfs2/

[33] A Fast File System for UNIX. Marshall K. McKusick et al., ACM Transactions on Computer Systems 2, 3, 1984

[34] The Design and Implementation of a Log-Structured File System". Mendel Rosenblum and John K. Ousterhout, Proceedings of the 13th Symposium on Operating Systems Principles (SOSP), 1991

[35] The Zebra Striped Network File System. John Henry Hartman, PhD Thesis, 1994

[36] Cooperative Caching: Using Remote Client Memory to Improve File System Performance. Michael Dahlin et al., Operating Systems Design and Implementation, 1994.

[37] Serverless Network File Systems. Tom Anderson et al., 15th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems, 1995.

[38] OceanStore: An Architecture for Global-Scale Persistent Storage. John Kubiatowicz et al., Architectural support for programming languages and operating systems (ASPLOS), 2000.

[39] Building a Reliable Mutable File System on Peer-to-peer Storage C. A. Stein, Michael J. Tucker, and Margo I. Seltzer SRDS 2002.

[40] A framework for evaluating storage system security. Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan, USENIX Conference on File and Storage Technologies (FAST) 2002

[41] FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. Atul Adya et al., 5th Symposium on Operating Systems Design and Implementation (OSDI) 2002

[42] Taming aggressive replication in the Pangaea wide-area file system. Yasushi Saito, Christos Karamanolis, et al. 5th symposium on Operating systems design and implementation (OSDI) 2002

[43] FUSE: Filesystem in Userspace, http://fuse.sourceforge.net/

[44] Kademlia: A Peer-to-peer Information System Based on the XOR Metric (2002). Petar Maymounkov, David Mazieres, IPTPS02, Cambridge, USA, March 2002.

Level of confidentiality and dissemination

By default, each document created within Grid4All is © Grid4All Consortium Members and should be considered confidential. Corresponding legal mentions are included in the document templates and should not be removed, unless a more restricted copyright applies (e.g. at subproject level, organisation level etc.).

In the Grid4All Description of Work (DoW), and in the future yearly updates of the 18-months implementation plan, all deliverables listed in Section 7.7 have a specific dissemination level. This dissemination level shall be mentioned in the document (a specific section for this is included in the template, both on the cover page and in the footer of each page).

The dissemination level can be defined for each document using one of the following codes:

PU = Public.
PP = Restricted to other programme participants (including the EC services).
RE = Restricted to a group specified by the Consortium (including the EC services).
CO = Confidential, only for members of the Consortium (including the EC services).
INT = Internal, only for members of the Consortium (excluding the EC services).

This level typically applies to internal working documents, meeting minutes etc., and cannot be used for contractual project deliverables.

It is possible to create later a public version of (part of) a restricted document, under the condition that the owners of the restricted document agree collectively in writing to release this public version. In this case, a new document code should be given so as to distinguish between the different versions.