Project no. 034567

Grid4All

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

Specification and proof-of-concept adaptation of a peer-to-peer file system (DFS) and VO-aware file system (VOFS) on a fully functional VBS.

Due date of deliverable: June, 2008.

Actual submission date: .

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

Organisation name of lead contractor for this deliverable: ICCS

Revision: final

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)	

Table of Contents

List of Figures

List of Tables

Abbreviations used in this document

Abbreviation/acronym	Description
DFS	Distributed File Services
VBS	Virtual Block Store, the peer implementing the Storage Service
MDDB	MetaData DataBase, the peer implementing the Filesystem Service
VOFS	Virtual Organisation-aware File System
PEP	Policy Enforcement Point
PDP	Policy Decision Point
PAP	Policy Administratino Point

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

1 Introduction

This deliverable documents the VBS+DFS architecture and presents it as a base for VOFS. It identifies further integration points of VOFS within Grid4All, and reports on design and implementation status on each one.

DFS is a federative peer-to-peer filesystem with a flexible storage layer, VBS. VBS and DFS were originally described as different systems but they were consolidated in one, VBS+DFS.

Each VBS+DFS peer serves files and storage resources accross a global overlay network. Distributed files can be linked together in a directory, distributed storage blocks can be assembled into a file or aggregated in storage allocation pools.

VOFS is built by deploying VBS+DFS in a Grid4All environment. The Grid4All environment considered consists of Semantic Store, VO Security and User Management, and Autonomic VO Management and Overlay Services. The deployed VBS+DFS is connected to those components and services and becomes VO-aware, an instance of VOFS.

Integration with Semantic Store and Security have been designed and implemented in prototype. Other integration points are in design phase, as their respective specifications and prototypes are implemented for Month 24.

This deliverable is affected but the deviation is documented in D3.2. VBS and DFS are not independent layers and have been developed as a single system we refer to as VBS+DFS. Since, the DFS architecture has already been introduced in D3.2 (instead of D3.3), its description in this deliverable is a revision of D3.2.

The VOFS part of this deliverable is closely related with developments in other work packages and the general effort for an integrated architecture. Therefore, designs and scenarios presented are not final and may represent different alternatives.

The document starts with a general architectural view of VBS+DFS in Section 2.

In Section 3, VOFS is presented as VBS+DFS in a Grid4All scenario.

Section 4, identifies and discusses the integration points introduced by the VOFS scenario.

Then proceeds, even if slightly overlapping, to present the detailed design and implementation in Section 5.

Sections 4 and 3 presents how the VBS+DFS fits in with the rest of Grid4All. Specifically, it describes how VOFS is built over VBS+DFS.

2 The VBS+DFS architecture

2.1 Overview

2.1.1 A Filesystem Network

The DFS+VBS layer is a network of peers that possess file hierarchies (or namespaces), file data and raw storage, and share them according to the authoritative policy set by their owners. Peers are instantiated as processes bound to a network address. Multiple peers may be run in a single machine.

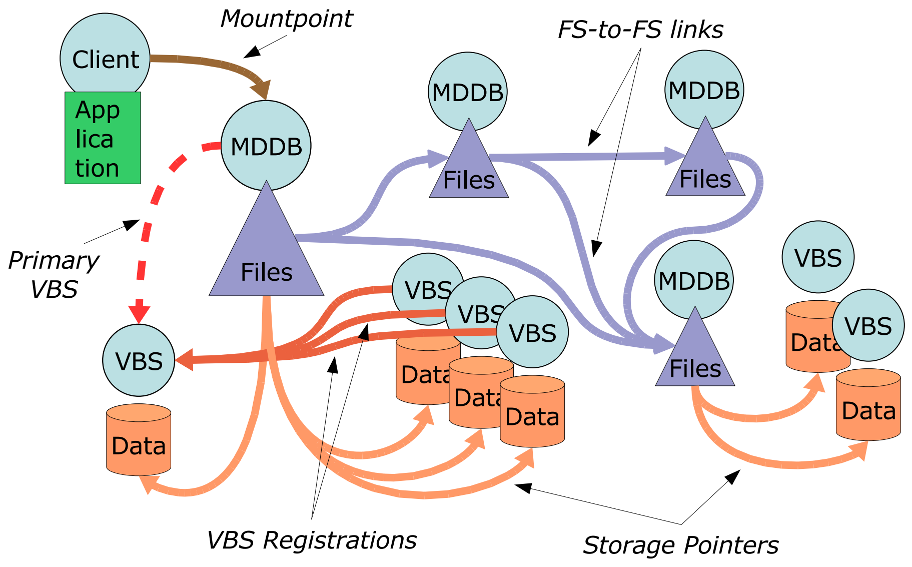

Figure 1: Schematic representation of a VBS+DFS network

The DFS+VBS layer is a peer-to-peer web-like network (fig. 1) where each peer serves its own file and storage. A peer may also possess links to foreign resources among its files. Linking works much like the links in Web pages. Foreign and remote resources can be made accessible from a single point by traversing those links.

2.1.2 Identity and Control

The identity of a peer is defined cryptographically by a public-private key pair by which peers can authenticate themselves. They can also protect their data and their communication by encryption. The owner of the identity is a physical person who may delegate authority to a software entity. Owners may set policy for access to the peer's resources.

Peers own local resources, which they completely control. A resource is globally identified by a URI (fig. 2) built from the identity of the peer owning the resource and a local identification. The local identification can be a file path or a random string that uniquely identifies the resource locally. URIs are references used to create peer-to-peer links.

Returning to the web page analogy, owners can shape their (peers') resources as freely as they can edit their webpage. Owners can also link to foreign resources, but as in the web, they cannot control the resources they link to. This is an important consideration for the DFS+VBS architecture:
Only an owner can add links to foreign resources amongst its local resources, while linked resources retain control over their resources. (Figure 3).

2.1.3 Resources

There are two types of resources; *filesystem* and *storage*. Filesystem resources are hierarchical file namespaces including (logically) the *content* of files. Storage resources refer to raw disk storage.

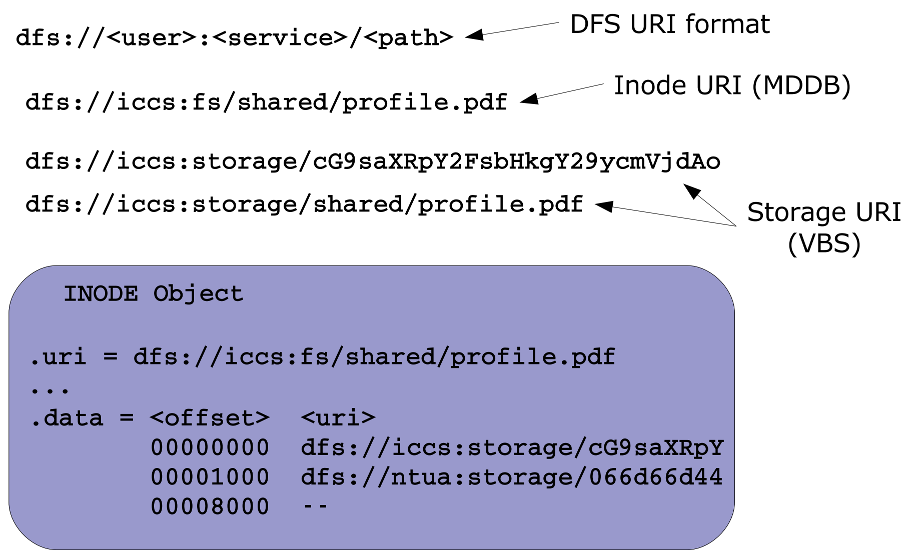
dfs://<user>:<service>/<path>
DFS URI format
dfs://iccs:fs/shared/profile.pdf
Inode URI (MDDB)
dfs://iccs:storage/cG9saXRpY2FsbHkgY29ycmVjdAo
dfs://iccs:storage/shared/profile.pdf
Storage URI (VBS)
INODE Object
.uri = dfs://iccs:fs/shared/profile.pdf
...
.data = <offset> <uri>
00000000 dfs://iccs:storage/cG9saXRpY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

Figure 2: Schematic representation of a URI construction

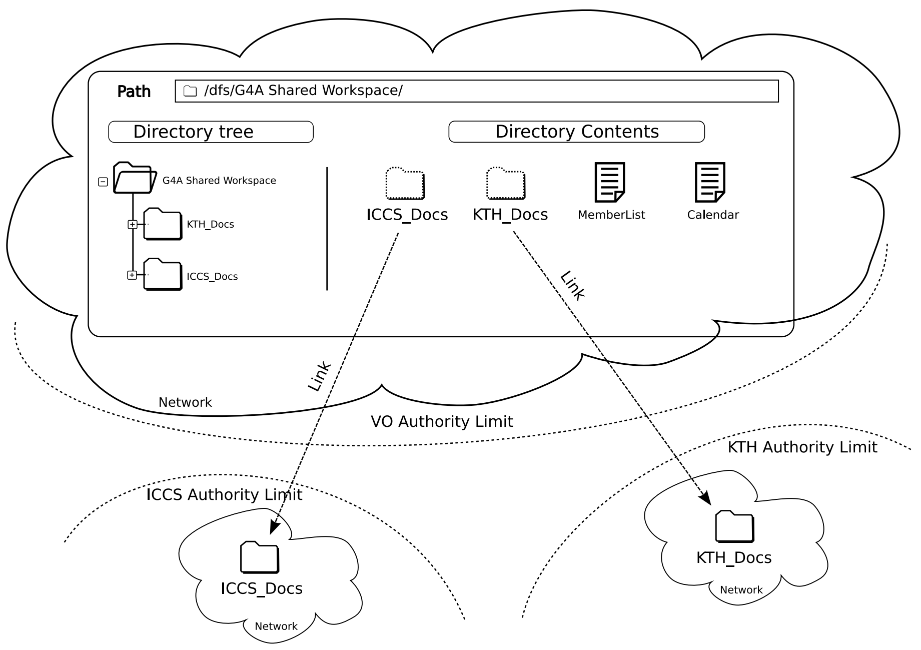

Figure 3: Federated resources remain under the control of their owners and the owner of the namespace controls federation

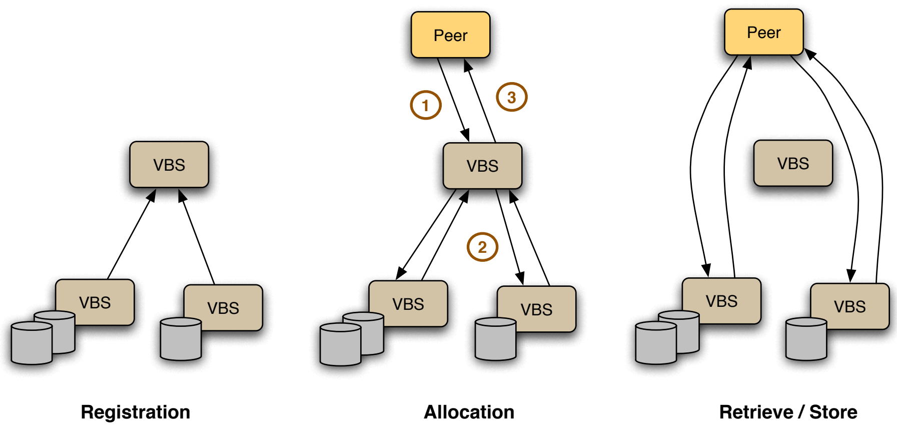

Figure 4: VBS registration and storage allocation

Requests to access resources must present *request tokens*. A request token always includes identification of both the *authority* that owns the resource, and the *agent* that requests the access. Resources are accessed at communication endpoints called *Services*. A service in the context of VBS+DFS may either be a Filesystem Service or a Storage Service.
A resource URI encodes the cryptographic Identity of the authority, the authority service and a path or handle for local identification (for filesystem or storage resources, respectively).

Request tokens can be filtered through a pluggable external filter, a Policy Enforcement Point (PEP), which communicates with its own policy service to provide authorisation (see Section 4.4). Users may associate their credentials as extended attributes to files (see Section 5.2.5), which are then included in the request tokens filtered through the PEP. The PEP privately uses any credentials provided.

2.1.4 Federation

Federation occurs at two levels. Aggregation through cross-peer *filesystem links* logically brings remote files in a local directory. Similarly, remote Storage resources can be logically joined together into a single, aggregated virtual image, a Storage Pool. When creating or extending files, storage blocks are allocated from such pools. Every filesystem service is associated with a *primary storage service*. Storage can be offered exclusively to a peer for its own filesystems by registering storage resources to the specific peer's primary storage service (fig. 4). The storage service creates a *storage pool* with the contributed resources.

Storage allocation is distinct from storage access. Allocation is performed by the VBS, while access is requested by the VBS+DFS clients and is served by the actual *storage provider*. VBS is the native storage provider but the architecture allows pluggable modules that implement allocation and access for different storage systems, such as HTTP, FTP, BitTorrent, or Grid-specific ones (see Fig. 5).

As client peers navigate through the network of links, they may cache remote resources. Caching is controlled by policies set by client owners. Caching allows access to resources even when network connectivity to remote peers is unavailable due to dis-

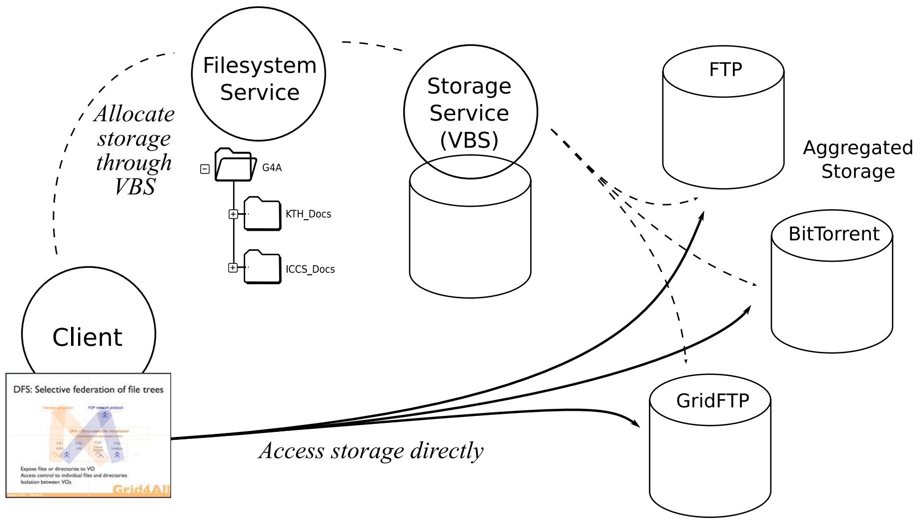

Figure 5: VBS allocation of storage and storage access

connection of either side. Also, attempts to access remote resources can be configured to be retried after network recovery or application restart.

DFS peering mechanisms include primitives for publish-subscribe communication and remote event notification. The mechanisms are used to handle the significant asynchrony in the network, which is further amplified by disconnected operation. Through the DFS Client interface, applications can also access these primitives.

2.2 Components

2.2.1 Filesystem Service (MDDB)

The Filesystem Service manages filesystem resources; file hierarchies and corresponding metadata. An entry, called *Inode*, in the file hierarchy may be a normal file, a directory or a peer-to-peer link. File inodes contain a list of storage URIs that point to the actual file data location. Inode metadata include general file properties and access policy. The set of maintained metadata is extensible.

The Filesystem Service also offers a publish-subscribe-based mechanism to send and receive file related events. Events may be either system-generated, such as file creation, modification, deletion, or general messages published by clients. Events are forwarded to a list of subscribers maintained in the inode metadata.

There may be many paths in the network to a given destination file, since many peers on the network maintain links towards it. The path to a file is marked by the identity of all peers encountered while traversing links. This enables file owners to authoritatively appoint foreign files into specific roles (places). Clients must traverse this specific *authoritative path* to retrieve appointed files. For example, an owner may link to a document describing where is his next meeting, but the document itself may belong to another. A client would have to go through the specific owner's link to correctly identify the next meeting place.

The Filesystem Service does allocate storage for file data. The clients are expected to find and allocate the necessary storage. Files are created by putting the data URIs in the file's inode, via the Filesystem Service. Nevertheless, each Filesystem Service may specify a *primary Storage Service* to accept storage contributions and to aid clients in allocating storage for their files.

2.2.2 The Storage Service: Virtual Block Store (VBS)

The Virtual Block Store (VBS) server implements the Storage Service, which is the managing authority for storage resources. The Storage Service accepts requests for storage allocation, returning URIs pointing to allocated storage. The URIs can be used directly as raw block storage, or can be put in a filesystem service inode metadata and be made part of a file.

The Storage Service is in fact an *aggregating* service. It is Storage Providers that own and provide storage to the Storage Service. Providers register storage (that they are willing to serve) at Storage Service. The Storage Service aggregates contributions into a pool and uses this pool to satisfy allocation requests. Once some storage has been allocated by the storage service, clients may access it directly through the storage provider. The storage provider identity is encoded in the storage URI. The URI also includes the storage access protocol (such as FTP, etc). VBS by default acts both as Storage Service and Storage Provider for itself. The VBS protocol is described in the previous deliverable D3.1/D3.2

Two basic storage allocation strategies are considered. The first iterates over a list of providers and the second traverses a prioritized queue of providers. Priorities are given to the system along with the storage resources.

2.2.3 The Storage Provider

The Storage Provider is a server that is not necessarily part of the DFS+VBS layer. The VBS server implements both the Storage Service and the Provider components, but the Provider maybe an external system with its own protocol, as long as clients are able to communicate with it.

2.2.4 The Client Service

The Client Service is a DFS peer along with a program that interfaces applications, making resources offered in a DFS network available to the local system. The client handles Filesystem, Storage and Provider protocols, walks Filesystem URIs by contacting Filesystem Services and following links, allocates and accesses Storage URIs for files, and subscribes to events or publishes to other subscribers — all as described in the previous sections. Clients also cache remote resources (inodes and data) and provide disconnected operation by serving requests from the local cache. Behaviour while disconnected is an important aspect of client functionality.

Cached resources at client side are governed by a protocol that handles serving file access requests and replica synchronisation in such a way that users perceive no disruption from the lack of connectivity. The protocol is defined via transitions in a number of states maintained for each cached file. Figure 6 documents these transitions. Terminology for state transitions is defined in table 2.2.4.

Term	**Definition**
read, write	refer to local client reads and writes.
downloading	acquiring a fresh copy of the cached (or not-yet-cached) resource.
uploading	updating the master copy of the cached resource with the local changes.
commit	update cache contents and state after a successful upload or download.
notification	is any message that implies the remote peer is online.
read while stale	means to read a file that is in the stale state.
	are several *timeouts*, triggering automatic state transitions, safeguarding the liveliness of the system in absence of activity.
	Any transition may be forced by policy logic.

Table 1: Terminology for state transitions

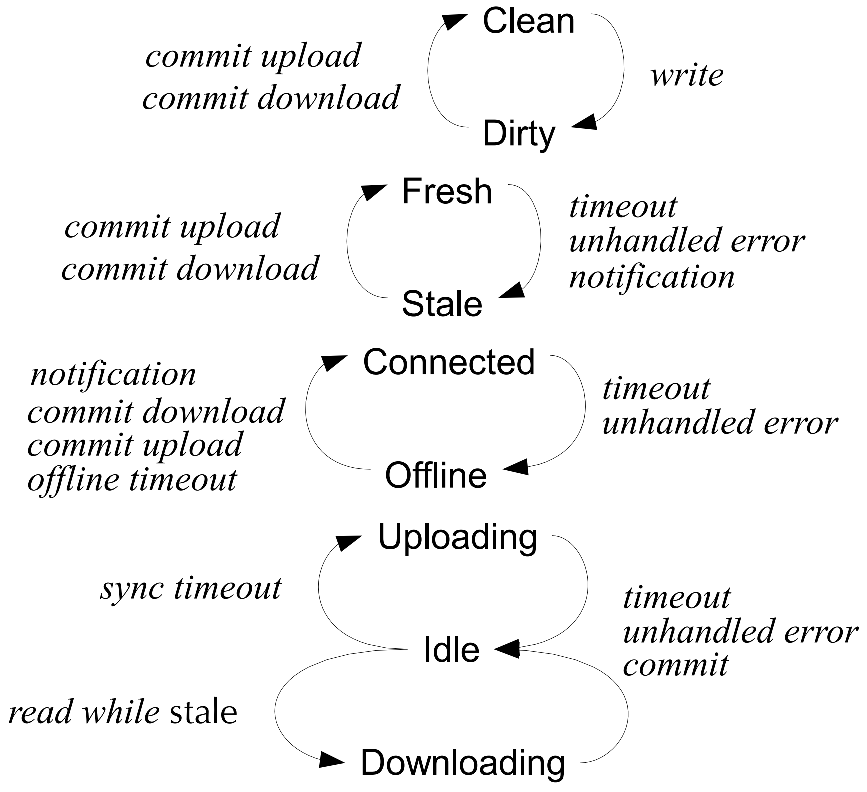

Figure 6: State transitions for cached resources

2.2.5 The POSIX Interface to the Client

The DFS+VBS layer includes a POSIX application interface to the Client, in the form of a mountable view of the DFS Network. This mountable view is rooted at a specific Filesystem Service, and paths below the mountpoint are translated as paths within the root Filesystem Service. The application, however, is not restricted to this root. Specially constructed paths within the mountpoint directly translate Filesystem URIs (see Section 5.5.1). Therefore any location in the DFS Network is accessible, regardless of the mounted root.

Other special paths to virtual files, provide access to DFS-specific configuration, such as the disconnected operation, and to facilities such as subscriptions, publications and notifications. Additionally, applications may associate configuration and private key-value data with any path within the mountpoint by reading and writing virtual files. These data items are kept persistently within the Local Client Store, as opposed to the Network Client Store, which caches remote resources. Data items for a specific path, if non-existent, may be inherited from parent paths.

3 VOFS on DFS+VBS

In a Grid4All scenario, VO-awareness of VBS+DFS has three main aspects: i) federation of filesystems, ii) VO access policy compliance, and iii) deployment and resource management.

The following subsections present scenarios describing how these aspects are introduced to a VBS+DFS network to introduce such VO-awareness. We refer to such a VO-aware VBS+DFS as VOFS.

3.1 Federation: Global VO Namespace

VOFS federates filesystems and storage resources by providing a *global VO namespace*. The global namespace is created as a DFS filesystem and may be administered and controlled by the VO administration. Each member of the VO is entitled to a home directory. Members may export files present in their own filesystems to their VO home directory. The global namespace may be further populated with other files. This depends on the requirements of the VO and is managed by the VO administrators. The global namespace may be populated with files exported to the VOFS, files newly created or shared documents (see Section 4.8 for the specific semantics of shared documents).

Federation of filesystems is technically implemented using the DFS mechanism to build cross-filesystem links (see Section 2.1.1). Storage pooling of contributed resources is handled by the *primary storage service* of the global namespace (see Section 2.1.4). Storage resources may be leased by trading at markets (see Section 4.7) and managed by the core resource management framework. The core resource management services are described within D2.2.

The global VO namespace scenario is demonstrated in Figure 7.

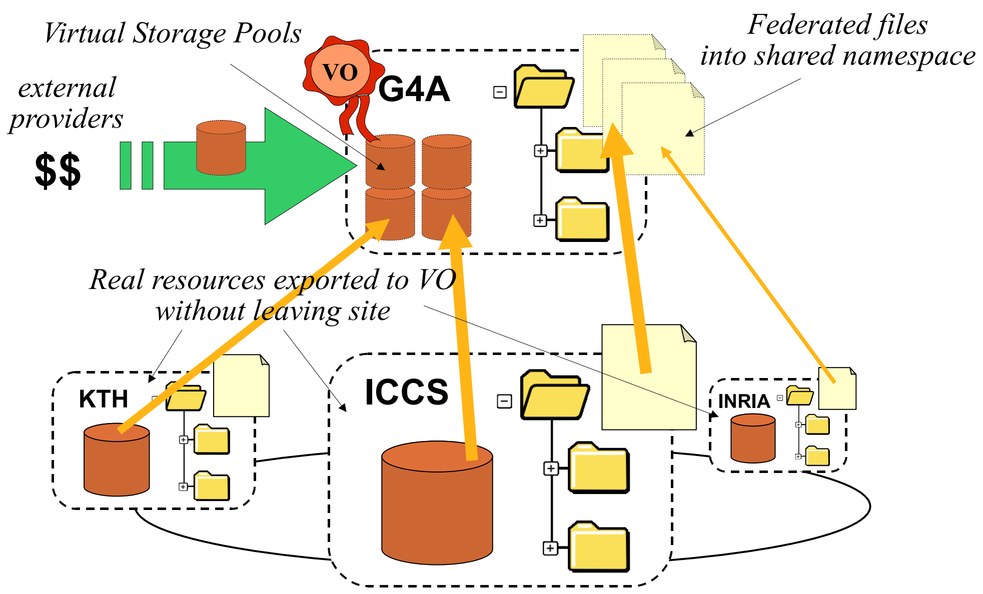
Virtual Storage Pools
external providers
$$
VO
G4A
Federated files into shared namespace
Real resources exported to VO without leaving site
KTH
ICCS
INRIA

Figure 7: Global VOFS namespace scenario using VBS+DFS federation features.

3.2 Policy: Policy Enforcement Points

The Grid4All security infrastructure ([01], Section 4.3.4) focuses on controlling access to resources according to policies which can be administrated independently of the resources. An important security component is the Policy Enforcement Point (PEP) through which each access request is filtered. Such a PEP is installed within the VBS+DFS services (see Section 4.4).

Policy creation and policy decisions are not handled by VBS+DFS. For decisions, a Policy Decision Point (PDP) is contacted through the PEP. Policies are authored using Policy Administration Points (PAP), by users or administrators.

User policies and credentials are stored in the local namespace (see Section 5.2.5) during initialisation or after user commands. When needed, they are pushed through the PEP as opaque objects.

3.3 Management: Configuration, Deployment

VOFS has to be included in the Grid4All middleware if its services are required for a VO. Part of the deployment is the initialisation of a global VO namespace and the provision of security information, such as Policy Decision Points (PDPs) and member credentials. See Section 4.4 for more on security policies.

There are two types of VOFS deployments, one for VOs and one for users. From the DFS point of view, VO peers are like any other. The main difference lies in additional configuration for creating the VO-wide workspace and initialising the VO-wide storage pool.

During VO lifetime, the VO peer has to be notified of important events such as members joining a VO or storage resources joining or leaving. When a member joins, its presence in the VO-wide workspace is set-up as a *home directory* under the new members control. The VO-wide storage pool grows or shrinks according to joins and leaves of offered storage resources.

The VBS layer is easily extensible and allows deployment of different underlying storage systems with minimal effort. Therefore, adminstrators may choose and deploy specific storage technologies in order to benefit from their performance or manageability properties.

4 Grid4All integration and connection with other Tasks

The DFS+VBS layer interfaces with the rest of the Grid4All architecture in these points:

1. The POSIX Client Interface, for applications and the Semantic Store.
2. The PEP authorisation module for external VO policy management.
3. The VBS Storage Provision layer, for external VO storage.
4. The Global Identity Index for locating users' peers in the network.
5. The inteface with the VO management for configuration, deployment and self management.

4.1 The POSIX Client Interface for applications

Grid4All applications may use the POSIX client interface for navigation, file create and storage, aggregation of distributed files into shared workspaces by linking, etc. as described in Section 5.5. Applications also benefit from the disconnected operations, file notifications and published message to subscribers. PEP credentials can be provided through the Local Client Store, as described previously.

4.2 The POSIX Client Interface for Semantic Store multilogs

Semantic Store provides support for concurrent and distributed editing of shared documents. The shared document called *multilog* has a special structure. It consists of log files and other files per editor. Each editor writes to its own log and reads from all others. Editors broadcast messages to all (live) editors to handle conflicting accesses to the shared document. Disconnected editors can continue to modify their local copy and modifications will be propagated when the editor reconnects.

Using the POSIX interface to the local client service, the Semantic Store can create a multilog document hierarchy which aggregates with links all the different editors' files. Each multilog instance also includes files private to each editor that need not be shared. Subscriptions provide notifications for new or modified log files. The broadcast messaging is handled by a publish-subscribe interface. Both notifications and publish-subscribe messaging are accessed through the POSIX interface (details in Section 5.5.2)

4.3 The Global Identity Index

Peers have unique identities that are related to their owners cryptographic certificates as members of a group or VO. These cryptographic identities are part of resource URIs, as described in Section 2.1.3. VOFS needs to translate a peer's identity into two pieces of information: The network address of the peer, and a cryptographic certificate for authenticating the peer's identity. VOFS expects this functionality implemented by the *Global Identity Index* module. This module is easy to implement on top of several established technologies, such as a trusted centralised database, a hierarchically distributed database (e.g. DNS), or a DHT. Peers always authenticate themselves in their communications so that no peer can fake its identity.

In a Grid4All setting, peers will use the VO membership service as the global identity index.

4.4 Policy Enforcement Point for VOFS

DFS+VBS peers (Filesystem, Storage, Client) authorise requests before they execute them on their resources. A request token always includes the identity of the requesting agent and the full URI of the target resource. The access is described with a type, a verb, a list of text arguments and a container of relevant serialised objects. There are several authorisation checks that can be enabled on resources. To enable external policy management required by VOFS (see Section 3), a call to an Policy Enforcement Point (PEP) is included as one of the authorisation checks. The PEP, after processing external to VOFS, responds with a positive or negative judgement, or a null one if the check is not applicable. For details see Section 3.2. ([01], Section 4.3.4).

Since PEP authorisation (and authentication) is external, the requesting clients must provide redentials for the PEP. These credentials are provided directly by the application and are used by the VOFS peer in its transactions. Applications may also need to provide different credentials depending on the authority they contact.

PEP credential collection is handled through the local namespace of the client (see Section 5.2.5). The application stores the credentials in pre-defined keys which are associated with the access controlled paths. The key-value pairs are inherited from parent paths, reducing the cost of administration and storage of credentials. On the other side, when the PEP is called during authorisation at the authority, the PEP is given the request token along with any PEP credentials available.

4.5 Market-Provided Storage

The VBS pools storage resources and manages their allocations through the Storage Service. Storage may be voluntarily donated to a specific VBS pool by VO members. In this case, the donating member selects the destination VBS pool. Storage applications may also request explicit allocation of storage resources to the Storage Service implemented by an instance of VBS. This could be done using the core VO resource allocation service.

In the case where applications are prepared to lease resources from the resource market-place, applications are expected to use interfaces of the Reservation Manager (documented in D2.2) to allocate storage and computational resources. Applications should specify the quantity of storage, the number of storage units, the time-frame within which they require the storage and a budget. The Reservation Manager will *callback* the application when leased storage resources join the VO.

4.6 WP1,WP2: Grid4All overlay infrastructure

The integrated Grid4All infrastructure based on WP1 overlays and component framework and on WP2 autonomic management is the basic environment for a VO. The infrastructure membership services will provide a global identity index for the VOFS (see Section 4.3).

4.7 Market Based Resource Management

Task 2.2 develops a market based resource management infrastructure. Storage traded in these markets can be allocated on behalf of users and registered to the VBS layer for usage. If storage bought in markets enters the Grid4All resource management infrastructure discussed in Section 4.6, then VBS can use this channel to take advantage of storage markets.

D2.3 has described the Grid4All market-place architecture. Leases to computational and storage resources may be allocated at the resource market-place. In a previous section, we have described that users should explicitly register storage providers to the Storage Service. Such registered storage is aggregated in storage pools and managed by its managing Storage Service. This mechanism permits smooth integration of storage resources of different provenance. Users may register their own storage or may also register storage that they have leased at resource markets.

A specific issue that needs to be addressed is that storage resources that are registered at the pool may be available only for limited time durations. When leases are about to expire, data stored in the expiring blocks will need to be restored elsewhere within the storage pool maintained by the Storage Service.

4.7.1 Data relocation for storage withdrawal

An important point is that in markets, storage is leased and therefore VBS must support a graceful withdrawal of storage with no loss of data. Upon notification of lease expiration the Filesystem Service responsible for the affected files must allocate new storage and relocate the data. This operation should be available to the users as well. The functionality can readily be implemented in a management logic unit with existing VBS+DFS mechanisms.

4.8 T3.3: Semantic Store

Task 3.3 enhances regular file storage with semantic functionality that allows applications to define constraints within and across documents and then manage conflicts that occur from the shared use of these. Semantic Store uses DFS to implement a special file structure, the *multilog*, that will represent those documents.

Semantic Store integrates with VBS+DFS. For details regarding the API, see Appendix A

4.9 WP4: Applications

Applications require storage for files in a distributed environment, possibly with the option to automatically search and allocate storage resources. Work Package 3 offers distributed storage of regular files through VOFS 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 VBS+DFS.

The simplest application scenario is global visibility of files. One instance of the application creates a file that is then opened by another instance in a remote machine (e.g. the *gMovie* application).

Applications like *e-Meeting* may use VOFS workspaces. In the *e-Meeting* scenario, a tutor uses one workspace to collect and organize all his teaching material, his *library*, and each virtual classroom uses each own workspace for material pertinent to the class. The tutor is able to select material from his library and instantly populate classroom workspaces using VOFS federation features. The tutor is able to control access to his library, classroom workspaces and is able to allocate storage resources for classroom assignments.

Finally, applications using Semantic Store, implicitly use VOFS via *multilogs*, the special Semantic Store documents.

5 Design and Implementation of the VBS+DFS prototype

5.1 Peering

We use the term *peer* to designate nodes that participate in the DFS and/or VBS infrastructure. DFS and VBS use peer-to-peer protocols to implement their services. The protocols and procedures for peer-to-peer communication in VBS+DFS are referred to with the general term *peering*. The following subsections describe the main peering mechanisms.

5.1.1 Communication Sessions

Peers establish stateful sessions in their communications to optimise performance, resource control and security. When required by policy, sessions are encrypted and authenticated.

Inter-peer communication sessions are encrypted, authenticated and authorized if the security policy demands this. To optimize the cost of session establishment, stateful sessions are maintained between peers.

Cryptographic support is in prototype status.

5.1.2 Request Token Routing

Request to access resources should present a *request token*. This object includes a pair of peer identities: the owning *authority* and the *agent* who requests access to the resource. Therefore, every resource access is attributed to a specific peer. The VBS+DFS peering mechanism delivers the request token to the recipient authority, taking care of URI translation and network connections. Replies to requests are also request themselves, and are routed through the same mechanisms back to the requesting agent, where the replies are matched with pending requests.

5.1.3 Consistency Considerations

Each resource is owned by an authority which serves the authoritative (or master) version of the resource. Requests are serialised by the serving authority. Sequential numbering of resource versions help to detect conflicts when an update based on a non-current copy is attempted. In a case of such a conflict, the users have to resolve it with external coordination. The Semantic Store layer of WP3 provides powerful functionality to address collaborative access to shared resources.

5.2 Storage and Caching

5.2.1 The Local Store Component

The VBS+DFS layer includes an internal database module where arbitrary objects can be persistently stored and retrieved by a key. The next sections describe functionality that is built on top of this local store. For implementation details see Section 5.7

5.2.2 Storing files

Files in the VBS+DFS network are data hosted on storage resources, along with metadata concerning the file. The URIs of the storage resources are considered part of file metadata. The file hierarchy is part of metadata as well. File metadata are stored internally within VBS+DFS peers. File data are stored in the local filesystem as normal files under the control of the VBS+DFS process.

5.2.3 Primary Resource Storage

The local store component of the VBS+DFS infrastructure is used by owner peers to keep their resources, either file data or file metadata and hierarchy, persistently stored and locally available.

5.2.4 Local Caching of Remote Resources

Remote resources, either file data or metadata, are never accessed directly by a client application. Instead remote resources are first locally replicated in a cache, and then accessed from cache. The VBS+DFS infrastructure attempts to keep those replicas synchronised with their source, either by pulling fresher versions or pushing locally updated versions. For consistency considerations see Section 5.1.3.

5.2.5 The Local Namespace

Distributed VOFS files are available locally to VOFS client machines at paths below the VOFS mountpoint. The VBS+DFS infrastructure allows users to create extended attributes to those files that are available only locally. The attributes are inherited from the parent paths. This allows for flexibile configuration while greatly reducing storage overhead and management complexity. An example usage of this namespace can be found in the authorisation procedure with an external policy enforcement point as described in Section 4.4.

5.3 Services

Services represent network endpoints that have authority over a certain type of resource: *filesystem*, *storage* or *client* service.

5.3.1 Filesystem Service (MDDB)

Filesystem Service peers implement the filesystem resource. Using the MDDB protocol, they serve file metadata including the hierarchical structure and the list of storage resources holding file data. It is also the point where subscriptions to events are kept and where notifications are generated and dispatched to subscribers.

Every user can create a filesystem service by creating an identity for it. Filesystem resources, designated with URIs, have a unique filesystem service as an owner. The owner's identity is encoded in the URI itself.

As a convenience to clients, each filesystem service is associated with a primary storage service, to handle storage management when clients cannot or will not do themselves. In the common course of events, clients obtain the primary storage service from the filesystem service itself. Another important function of the primary storage service is to accept contributions intended for a specific filesystem or his owner. Registering contributions facilitates allocation, but does not enforce any policy. Policies are enforced during any actual resource access.

5.3.2 Storage Service (VBS)

Storage services manage storage aggregation and subsequent allocation from the resulting storage pool. The native VBS storage service will also serve the storage for read or write access by clients. However, the actual serving of the storage is independent from the allocation so that varying types of storage providers may be used (e.g. FTP, HTTP, gridFTP, etc.)

A storage service may also act as a Storage Provider to another Storage Service Storage offered to a pool is treated as allocated storage by the donor, while further management is up to the Storage Service managing the pool.

5.3.3 Client Service

Client services represent the working environment of users, including their files and the applications accessing them. As a peer, the service communicates with filesystem, storage services and storage provides to retrieve all kinds of filesystem data. The client service also subscribes for asynchronous notifications of modifications or user publications taking place on remote resources.

5.4 Operation and Usage

This section generally describes the various capabilities of the system as well as its intended usage. For specifics about the POSIX interface see Section 5.5.

5.4.1 Connected Operation

In normal operation, that is, when all VBS+DFS peers are connected, the VBS+DFS interface behaves like a traditional filesystem. The only abnormal situation is conflicting concurrent modifications from several hosts to the same file. For more details see Section 5.5

5.4.2 Event Notifications

Clients may subscribe to modifications happening to specific files accross the network, including changes in file names and hierarchy. There is a special PUBLISH event that is triggered explicitly by clients, used to implement per file publish-subscribe services.

5.4.3 Disconnected operation

Disconnected operation is possible owing to the local cache of each client. While disconnected, cached files remain available for read and write access but are not synchronised with the authoritative sources. When the authority of a file cannot be contacted within a certain timeout, the *access timeout*, the client considers itself disconnected from this peer and all files of this authority gain the *offline* status. While accessing offline files, no network access is attempted. This is necessary because otherwise every access to an offline file would have to wait for the network timeout. However, after an other timeout occurs, the *offline timeout*, the file loses its offline status and subsequent access will result in contacting the authority over the network. To enhance flexibility, all timeouts are configurable and files can be forced in and out of the offline status explicitly.

The local cache uses an LRU policy for 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.

5.5 The POSIX interface to the Client

This section presents details about the POSIX interface to the client service of the VBS+DFS system. The semantics of (a sufficiently large subset of) the POSIX API are extended in an non-intrusive way in order to support an interface to the functionality specific to the VBS+DFS. VBS+DFS is mountable in the local system like conventional filesystems. While the system has a valid POSIX behaviour, applications may encounter errors more frequently.

5.5.1 Virtual Files

Virtual files are special files that do not represent a DFS file, but their contents and metadata are dynamically computed. They are artifacts that have been introduced to support DFS filesystems in a POSIX-compliant way.

In order to preserve compatibility, virtual files are normally invisible to directory listings. Nevertheless, they remain accessible, as if someone creates them the moment before the access and deletes them the moment afterwards (this is valid behaviour in conventional filesystem).

Paths to virtual files contain the reserved character '`@`'. This character cannot be used in normal filenames. The character '`@`' splits the path into two distinct paths:

```
<real path>@<virtual path>.
```

This way, every virtual path is associated with a real one.

The most important virtual file is named '`@`' and it is present (linked, following the conventional terminology) in every directory. The real path is ignored and its virtual component is the empty string. It represents the Global Identity Index, as it allows access to any peer in the network. By convention, the paths of the form '`@/`*URI*' are redirected to the filesystem service with identified by the *URI* component. Therefore, cross-filesystem links are created as symbolic links to targets within this global virtual file:

```
ln -s @/Grid4All:fs/vo/workspace link-to-g4a-workspace
```

5.5.2 Subscriptions and Notifications

POSIX operations on files and directories within a file system hierarchy can be signalled to remote peers.

Subscriptions are individually maintained but the in POSIX interface they are grouped by the *notification endpoint*. A notification endpoint is a virtual file where notifications appear.

Notification endpoints can be created by writing a configuration script to the global virtual file '`@subscribe`'. [1] The endpoint appears in the global virtual directory '`@notify/`', while the configuration is accessible in an identically named file within the global virtual directory '`@subscriptions/`'. The configuration script lists the name of the endpoint and a list of specific subscriptions to add or remove. A subscription specifies a file and an event name. Publications can be made by writing to the virtual file '`@publish`'. Notification endpoints can be deleted by deleting their configuration files in the '`@subscriptions/`' directory.

Processes that open the notification endpoint will be signalled with a POSIX SIGIO signal whenever a new notification arrives. Table 5.7 lists the subscribable events. Section A.5 details subscription configuration.

5.6 Manipulating the filesystem

DFS+VBS file systems may be manipulated in the same way as any POSIX file system. However, special procedures are required for storage allocation and cache management.

[1] Global virtual files are independent of the real path; they are the same for all files

Usage	Software	Version	Comments
Implementation platform	Python	2.5	extensions written in C
POSIX interface	FUSE	2.7	Linux and OS X
Persistent store	Berkeley DB	4.7	

Table 2: Software Requirements for VBS+DFS

5.6.1 Cache Management

Caches maintain local copies of remote files and contents. Caches are monitored and local copies are synchronized with their remote sources automatically. Locally modified files are written back, or newer versions are pulled back from the remote source.

Cache management is based on a state transition system for every file in local cache. There are two independent binary states:

- *clean* if it has not been locally modified or *dirty*, if it has
- *fresh* if it is considered to be up-to-date with the source, or *stale* if considered otherwise.

Dirty files, are queued to be synchronised with their sources. Synchronisation is started at regular time intervals.

Clean and dirty state transitions are due to local events and hence are easy to manage. *Stale* resources require communication with the remote source. DFS makes no assumptions about network connectivity at any time, so it follows a best effort approach. Stale resources are served immediately, but a background download is launched to check for new versions. When a file is synchronised, it is marked as *fresh* and a new timeout is armed. An access after the expiry of this timeout, causes the file to be compared with its source for freshness. Asynchronous notifications may also force files into *stale* status, regardless of timeouts.

Section 5.4.3 describes operation when a client node is disconnected from its peers. Conflicts are detected when a file is both *dirty* and *stale*.

5.6.2 Storage Allocation

Clients are the ones that are responsible to allocate storage. To shake that burden off unsophisticated or unwilling clients, each filesystem is associated with a primary storage service that is normally willing to offer storage for files (see Section 2.2.1). Therefore, the simplest solution to storage allocation is for each client to turn to the primary VBS of the filesystem it has mounted, or the primary VBS of the filesystem where the new file will be created.

Allocation customisation, that is, which VBS to contact, can be set up on a per file basis, using the local namespace (Section 5.2.5).

5.7 Technology Used and other Implementation Details

This section presents details about the implementation of the VBS+DFS prototype.

Event	Description
`CREATE`	A file was created in the subscribed directory
`REMOVE`	A file that was deleted from the subscribed directory
`RENAME`	A file that was moved from the subscribed directory
`MODIFY`	Data or metadata of the subscribed file were modified
`PUBLISH`	A user published a message on this file

Table 3: Description of subscribable DFS events

5.7.1 Persistent Store Component

This component has a generic modular interface to a low-level storage backend that supports transactions. Currently Berkeley DB is used as the back-end, but can be replaced with minimal effort.

5.7.2 Local Namespace

The local namespace is actually a very generic interface to the system. One can review the entire internal system state at any time and other than system-specific keys, users can enter arbitrary key-value pairs. This namespace is also available through the extended file attributes filesystem interface.

A Semantic Store interface with VBS+DFS

A.1 The DFS Virtual Files

The FUSE interface for the DFS extends normal filesystem semantics to include DFS-specific operations. DFS functionality is available using POSIX calls to special, dynamically generated files, called *virtual files*.

Virtual files are accessed with the use of a special character '`@`'. In every POSIX call that includes a filename, the absolute path of the file is partitioned into two segments at the rightmost '`@`' character. The first segment represents the *real* path and the second a *virtual* path which is associated with the real path given. If there is no '`@`' character in the path, then the path is *real*.

A.1.1 Reading and writing Virtual Files

The FUSE software intercepts system calls accessing the filesystem and then calls the underlying DFS libraries to serve them. However, the calls made by FUSE to the underlying DFS library do not correspond (e.g. in offset-length arguments) to the system calls that triggered them. System calls are translated by FUSE to page-filling requests, in way similar to how the operating system translates clusters of individual writes to page-sized disk writes. This significantly limits control over the POSIX behaviour of the file and reduces the flexibility for implementing virtual files. Therefore, most Virtual files are either read-only or write-only.

Virtual files have dynamic contents that are determined upon opening the file. Writes to a virtual file are not processed by DFS until the file is closed. In special cases, other mechanisms may be used that have the same effect as closing and reopening the file, such as seeking to the begining of the file.

A.2 The Telex requirements for Multilogs

Multilogs are documents in the form of a hierarchy of files, and are shared among many editors. Each editor only writes to his own files and reads from all other editors' log files. Using the peer-to-peer linking capabilities of DFS, log files can be aggregated under a single directory shared by all editors.

Telex needs two communication facilities from the DFS:

i. *Telex needs to be notified when a log file has been modified or was added or deleted.*

ii. *Telex needs to be able to broadcast private messages to all editors and receive such messages.*

A.3 A Java wrapper class

DFS Provides a wrapper Java class that implements this functionality by using virtual files in a special way. The interface for this functionality is:

Method	Description
`publish`	publish a message to subscribers of a path
`subscribeFile`	add a subscription for a file
`unsubscribeFile`	remove a subscription for a file
`waitForNotification`	block execution until any notification arrives
`update`	overridable callback for that receives notification

Table 4: Description of methods of the Java Telex-DFS interface

```
public interface IDFS
{
    void publish(String path, byte[] message) throws IOException;
    void subscribeFile(String path, String event) throws IOException;
    void unsubscribeFile(String path, String event) throws IOException;
    Vector waitForNotification() throws InterruptedException;
    void update(final Vector msg);
}
```

Applications must instantiate one or more wrapper objects per multilog.

Using the `subsribeFile()` and `unsubscribeFile()` methods, callers subscribe to events for specific files within the multilog.
The `waitForNotification()` method blocks until any subscribed notification for any subscribed file arrives. Specifically:

- To monitor for new or deleted log files, the Telex application must subscribe for '`CREATE, REMOVE`' notifications to the containing directory.
- To monitor for modifications to a file, the Telex application must subscribe for '`MODIFY`' notifications to that file.
- To subscribe for private messages to a file, the Telex application must subscribe for '`PUBLISH`' notifications to that file.
- To broadcast a private message to all subscribers of a file, the Telex application must publish the message to that file using the `publish()` method.

A.4 Format of notification Messages

The notification messages returned by the wrapper are composed of three fields, each delimited with a byte '`:`' (colon):

1. The first field contains a string identifying the event type (e.g. '`MODIFY`').
2. The second field contains a string identifying the path (relative to the DFS mount-point) of the file the notification was destined for.
3. The third field contains a binary message as payload for the notification, or one or more whitespace characters if there is no such payload.

Messages for publication are submitted to the wrapper as they are, with no headers or delimiters. The messages submitted will appear in the third field of the '`PUBLISH`' notification.

A.5 Low level virtual file access

In order to subscribe for notifications, an application writes a configuration script in the virtual file '`@subscribe`'. The text is delimited by newline characters. The first line contains the name of a notification file to be created. All notifications that are subscribed by this script will be delivered to that file. The notification file will reside in the virtual '`@notify`' directory, which is global; this same virtual file is accessible at any directory under a DFS mountpoint.

The remaining lines are of the form:

`[-]`⟨*EVENT*⟩`:file:`⟨*path/to/file*⟩

For example,

```
CREATE:file:/docs/calendar/logs
```

Subsequent writes to the same notification endpoint file (specified in the first line) will be applied on the contents already written by other writes. New lines cause new notification subscriptions to be added to the existing, while prepending a '`-`' to the line unsubscribes an existing one.

There are various methods of waiting for notifications to appear in the notification endpoint files. The safest and most compatible method is for the DFS client to send a POSIX signal ('`SIGIO`') to the processes that maintain an open file descriptor to the notification endpoint file. The wrapper Java class hides these details so the implementation may change.

A.6 Communication guarantees

Clients persistently cache resources, especially dirty ones. This persistent state allows retrial of communication for an unspecified period of time, until the relevant accesses are successful or aborted by the user. Therefore, while there are no guarantees from the network, communication is eventually guaranteed by retrial.

References

[01] "Grid4All: Democratic Grids — Scenario and Architecture". Grid4All Deliverable D4.6, 2008