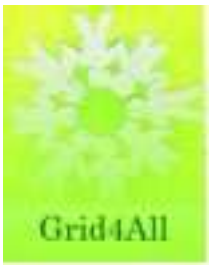


Project no. 034567

Grid4All

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

Deliverable 3.1
Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities

Due date of deliverable: June 2007.

Actual submission date: 20 June 2007.

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

Organisation name of lead contractor for this deliverable: INRIA Regal

Revision: 2007-06-20

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)	√

Preamble

This document, Deliverable 3.1 “Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities”, comprises the following parts:

Chapter I	Semantic Store
Chapter II	Collaborative Applications
Chapter III	VO-Aware File system
Appendix I	Telex application API

The pages of each part are numbered separately.

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.).

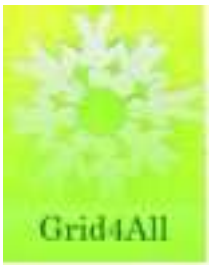
In the Grid4All Description of Work (DoW), and in the future yearly updates of the 18-months implementation plan, all deliverables listed in Section 7.7 have a specific dissemination level. This dissemination level shall be mentioned in the document (a specific section for this is included in the template,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



Project no. 034567

Grid4All

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

Deliverable 3.1: Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities - Chapter I

Due date of deliverable: June, 2007.

Actual submission date: June, 2007.

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

Organisation name of lead contractor for this deliverable: INRIA Regal

Revision: Submitted 2007-06-20

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

Abbreviations used in this document

Abbreviation/acronym	Description
ACF	Action Constraint Formalism
Grid4All	The FP6 STREP project that aims to democratise access to the Grid, through the use of peer-to-peer technologies. The co-ordinator is France Télécom; the other partners are a SME from Spain, and research labs from Greece, France, Spain, and Sweden.
VO	Virtual Organization
VOFS	VO-aware File System

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 the Chapter I of Deliverable 3.1 ”Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities”, which comprises the following parts:

Chapter I	Semantic Store
Chapter II	Collaborative Applications
Chapter III	VO-Aware File system
Appendix I	Telex application API

The following persons contributed to this chapter: Sarfraz Ashfaq INRIA-Atlas, Lamia Benmouffok INRIA-Regal, Jean-Michel Busca INRIA-Regal, Maria Gradinariu INRIA-Regal, Esther Pacitti INRIA-Atlas, Marc Shapiro INRIA-Regal, Pierre Sutra INRIA-Regal, Patrick Valduriez INRIA-Atlas, Martin Vidal INRIA-Atlas.

1 Introduction

Grid4All aims at supporting collaborative applications in a peer-to-peer (P2P) environment. Collaboration raises a key problem in distributed systems: access to shared and mutable data, and this problem is even more difficult in a P2P systems due to the volatility of the participants.

A possible approach is optimistic replication (OR). OR decouples data access from network access: it allows a processor to access a local replica without synchronising. A processor makes progress, executing uncommitted actions, even while others are slow or unavailable. Local execution is tentative and actions may roll back later. An OR system propagates updates lazily, and ensures consistency by a global *a posteriori* agreement on the set and order of actions.

However previous implementations of OR do not take into account the semantics of the collaboration. Therefore they are either incomplete, or inefficient, or both. For instance, many algorithms implement a total order, which is inefficient in the common case where many actions commute.

In our previous work, we brought semantics to OR and proposed the Action Constraint Framework (ACF), which is especially well suited for collaborative applications. In the ACF, a shared document is represented by a set of actions (or operations) submitted by users, and the set of constraints between those actions, expressing the semantics. A formal description of the ACF can be found in Grid4All Deliverable D1.3.

We are currently developing a middleware system, called *Telex*, that provides communication and consistency services based on this model. Developpers may focus on the core functionalities of their application, leaving the difficult issues of managing consistency to Telex. We propose to re-use and adapt Telex in order to implement the Semantic Store layer of Grid4All. Telex will enable collaboration in P2P environments, based on optimistic replication and semantic-aware agreement.

This chapter describes the Telex midddleware and its use within the Grid4All project. The chapter is organized as follows. Section 2 studies the state of the art on semantic replication and storage. Section 3 analyzes the requirements of the semantic store. Section 4 describes in detail the architecture of Telex and its operation. Finally, section 5 describes how Telex interfaces with the underlying VOFS/VOFS in Grid4All. The interface that Telex exposes to application is described in detail in Appendix I.

2 State of the art

This section describes the state of the art of data replication in P2P systems. A more comprehensive survey can be found in Martins et al. [30].

We first present an overview of data replication, within the larger area of distributed systems. Then we focus on the optimistic approach, which supports good properties for dynamic environments. Finally we expose data replication techniques in the particular context of P2P systems.

2.1 Data replication techniques

Data replication consists of maintaining multiple copies of data objects, called replicas, on separate sites [43]. An object is the minimal unit of replication in a system. For instance, in a replicated relational database, if tables are entirely replicated then tables correspond to objects; however, if it is possible to replicate individual tuples, then tuples correspond to objects. Other examples of objects include XML documents, typed files, multimedia files, etc. A replica is a copy of an object stored on a site. We call state the set of values associated with an object or a replica at a given time.

Data replication is very important in the context of distributed systems for several reasons. First, replication improves the system availability by removing single points of failures (objects are accessible from multiple sites). Second, it enhances the system performance by reducing the communication overhead (objects can be located closer to their access points) and increases the system throughput (multiple sites serve the same object simultaneously). Finally, replication improves the system scalability as it supports the growth of the system with acceptable response times. However, data replication in distributed system has a major issue: how to manage replicas, or equivalently how to manage updates.

Gray et al. [20] classify *replica control mechanisms* according to two parameters: where updates take place (i.e., which replicas can be updated), and when updates are propagated to all replicas to be executed. According to the first parameter (i.e., where), replication protocols can be classified as single-master or multi-master solutions, as described in sub-section 2.1.1. According to the second parameter (i.e., when), update propagation strategies are divided into synchronous (eager) and asynchronous (lazy) approaches, as described in sub-section 2.1.3. The replica control mechanisms are also affected by the way in which replicas are distributed over the network (replica placement). Sub-section 2.1.2 discusses the full and partial replication alternatives.

2.1.1 Single vs. multiple masters

A replica of an object can be classified as primary copy or secondary copy according to its updating capabilities. A primary copy accepts read and write operations and is held by a master site. A secondary copy accepts only read operations and is held by a slave site.

In the single-master approach, there is only a single primary copy for each replicated object. In this case, every update is first applied to the primary copy at the master site, and then it is propagated towards the secondary copies held by the slave sites. Due to the interaction between master and slave sites, this approach is also known as master/slave replication. Centralizing updates at a single copy avoids concurrent updates on different sites, thereby simplifying the concurrency control. In addition, it ensures that one site has the up-to-date values for an object. However, this centralization introduces a potential bottleneck and a single point of failure. Therefore, a failure in a master site blocks update operations, and thus limits data availability.

In the multi-master approach, multiple sites hold primary copies of the same object. All these copies can be concurrently updated, wherefrom the multi-master technique is also known as update anywhere. Distributing updates avoids bottlenecks and single points of failures, thereby improving data availability. However, and in order to ensure data consistency, concurrent updates to different copies must be coordinated and a reconciliation procedure must be applied to ensure the consistency between replicas.

2.1.2 Full replication vs. partial replication

Replica placement over the network directly affects the replica control mechanisms. Full replication consists of storing a copy of every shared object at all participating sites. This approach provides simple load balancing since all sites have the same capacities, and maximal availability as any site can replace any other site in case of failure.

With partial replication, each site holds a copy of a subset of shared objects, so that the objects replicated at one site may be different of the objects replicated at another site. This approach expends less storage space and reduces the number of messages needed to update replicas since updates are only propagated towards some sites. However, if related objects are stored at different sites, the propagation protocol becomes more complex as the replica placement must be taken into account. In addition, this approach limits load balancing since certain sites are not able to execute a particular set of transactions.

2.1.3 Synchronous vs. asynchronous systems

In distributed database systems, data access is done via transactions. A transaction is a sequence of read or/and write operations followed by either a commit or abort, if the transaction does not complete successfully. A transaction that updates a replicated object must be propagated to all sites that hold replicas of this object in order to keep its replicas consistent. Such update propagation can be done within the transaction boundaries or after the transaction commit. The former is called synchronous, and the latter asynchronous propagation.

Synchronous propagation The synchronous update propagation approach (a.k.a. eager) applies changes to all replicas within the context of the transaction that initiates the update. As a result, when the transaction commits, all replicas have the same state.

This mechanism is achieved by using concurrency control techniques such as two-phase locking (2PL) [47] or timestamp based algorithms. In addition, a commitment protocol like two-phase commit (2PC) [47] can be run to provide atomicity (either all transactions' operations are completed or none of them are). Thus, synchronous propagation enforces mutual consistency among replicas. Bernstein et al. [7] define this consistency criteria as *one-copy serializability*, i.e., despite the existence of multiple copies, an object appears as one logical copy (one-copy equivalence). Namely a set of accesses to the object on multiple sites is equivalent to the serial execution of these accesses on a single site.

Early solutions [5, 45] use synchronous single-master approaches to ensure one-copy serializability. However, most of the algorithms avoid this centralized solution and follow the multi-master approach by accessing a sufficient number of copies. For instance, in the ROWA (read-one/write-all) approach [7], read operations are done locally while write operations access all copies. ROWA is not fault tolerant since the update processing stops whenever a copy is not accessible. ROWAA (read-one/write-all-available) [6] overcomes this limitation by updating only the available copies. Another alternative are quorum protocols [17, 22], which can succeed as

long as a quorum of copies agrees on executing the operation. Other solutions combine ROWA/ROWAA with quorum protocols [16, 2].

More recently, Kemme and Alonso [26] proposed new protocols for eager replication that take advantage of group communication systems, avoiding some performance limitations of existing protocols. Group communication systems [10] provide group maintenance, reliable message exchange, and message ordering primitives between groups of nodes. The basic mechanism behind these new protocols is to first perform a transaction locally, deferring and batching writes to remote replicas until transaction commit time. At commit time all updates (the write set) are sent to all replicas using a total order multicast which guarantees that all nodes receive all write sets in exactly the same order. As a result, no two-phase commit protocol is needed and no deadlock can occur. Following this approach, Jiménez-Peris et al. [23] show that the ROWAA approach, instead of quorums, is the best choice for a large range of applications requiring data replication in cluster environments. Next, in [29] the most crucial bottlenecks of the existing protocols are identified, and optimizations are proposed to alleviate these problems, making one-copy serializability feasible in WAN environments of medium size.

The main advantage of the synchronous propagation is to avoid divergences among replicas. This enables local reads since transactions surely take up-to-date values. The drawback is that the transaction has to update all replicas before committing. If one replica is unavailable, this can block the transaction, making synchronous propagation unsuitable for dynamic networks such as P2P. In addition, the transaction response times and the communication costs increase with the number of replicas and, for these reasons, this approach does not scale beyond a few tens of sites.

Asynchronous propagation The asynchronous update propagation approaches (a.k.a. lazy) do not change all replicas within the context of the transaction that initiates the updates. Indeed, the initial transaction commits as soon as possible, and afterwards the updates are propagated to all replicas. Asynchronous replication solutions can be classified as optimistic or non-optimistic according to their assumptions concerning conflicting updates. In general, optimistic replication relies on the assumption that conflicting updates will occur only rarely, if not at all. Updates are therefore propagated in the background, and occasional conflicts are fixed after they happen. In contrast, non-optimistic asynchronous replication assumes that update conflicts are likely to occur and implements propagation mechanisms that prevent conflicting updates.

An advantage of the asynchronous propagation is that the update does not block due to unavailable replicas, which improves data availability. In addition, communication is not needed to coordinate concurrent updates, thereby reducing the transaction response times and improving the system scalability. In particular, the optimistic asynchronous replication is much more flexible than other approaches as the system can choose the appropriate time to propagate updates and the application can progress over a dynamic network in which nodes can connect and disconnect at any time. The main drawback of optimistic techniques is that replicas may diverge, and then local reads are not guaranteed to return up-to-date values. The non-optimistic asynchronous replication is not as flexible as the optimistic approach, but it provides up-to-date values for local reads with high probability.

Non-optimistic approaches The goal of non-optimistic asynchronous solutions is to use lazy replication while still providing one-copy serializability. Chundi et al. [13] have shown that serializability cannot be guaranteed in all cases. To circumvent this problem, it is necessary to restrict the placement of primary and secondary copies across the system. The main idea is to define a set of allowed configurations using graphs, so that nodes represent sites and edges

between sites represents links between primary and secondary copies of a given object. If this graph is acyclic, serializability can be guaranteed by simply propagating updates sometimes after transaction commits [13].

Pacitti et al. [32, 31] have enhanced these initial results by allowing certain cyclic configurations. The replication algorithm assumes that (1) the network provides FIFO reliable multicast, (2) an upper bound on the time to multicast a message from a node to node (noted Max) is known and (3) that local clocks are ε-synchronized i.e., the difference between any two correct clocks is not higher than ε). As a result, a transaction is propagated in at most $Max + \varepsilon$ units of time and thus chronological and total orderings are ensured with no coordination among sites. Experimental results show that such approaches ensure a consistency level equivalent to one-copy serializability for normal workloads, and for bursty workloads the consistency level is still quite close to one-copy serializability. Coulon et al. [15] have extended this solution to work properly in the context of partial replication.

Breitbart et al. [8] propose alternative solutions. The first one requires acyclic directed configuration graphs (edges are directed from primary copy to secondary copy). The second solution, in contrast, allows cyclic graphs, and applies lazy propagation along acyclic paths while eager replication is used whenever there are cycles. Since these approaches use lazy update propagation, the state of a replica can be somewhat stale with respect to committed (validated) transactions. Thus, the associated consistency criterion is freshness, which is defined as the distance between two replicas wrt. validated transactions.

Optimistic approaches Contrasting with non-optimistic approaches, optimistic replication does not aim at providing one-copy-serializability. Indeed, it assumes that conflicts are rare or do not happen. Thus, the propagation of updates is done in background and replica divergences may arise. Conflicting updates are reconciled later, which means that the application must tolerate some level of divergence among replicas. This is acceptable for a large range of applications, which are described in the following section.

2.2 Optimistic replication

This section presents several systems based on optimistic replication. In the description, we focus on features that differentiate Telex from those systems.

2.2.1 DNS

The Domain Name System [4] is the standard hierarchical name service for the Internet. Names for a particular zone (a subtree in the name space) are managed by a single master site that maintains the authoritative database for that zone and optional slave sites that copy the database from the master. The master and slaves can answer queries from remote sites.

2.2.2 LOCUS

LOCUS [33] is a distributed operating system composed by a replicated file system. The file system uses version vectors to order updates on distinct replicas of the same object. A version vector is an array of timestamps that allows the detection of conflicting updates. For LOCUS, any two concurrent updates to the same object are in conflict. LOCUS automatically resolves conflicts by taking two versions of the object and creating a new one.

2.2.3 TSAE

Time-Stamped Anti-Entropy [18] uses real-time clocks to order operations. Basically, sites exchange vector clocks (i.e., arrays of timestamps) and acknowledge vectors in order to learn about the progress of others, so that a site i is able to determine which operations have surely been received by all sites at a given time. As a result, site i can safely execute these operations in the timestamp order and delete them. TSAE does not perform any conflict detection or resolution. It only needs to agree on the set of operations and their order.

2.2.4 Ramsey and Csirmazs' file system

Ramsey and Csirmaz formally study the semantic of a simple file system that supports few operation types, including create, remove, and edit [39]. For every possible pair of concurrent operations, they define a rule that specifies how the operations interact and may be ordered. Non-concurrent operations are executed in the submission order.

2.2.5 Unison

Unison [36] is a file synchronizer that reconciles two replicas of a file or directory based only on the current states of the replicas (i.e., it does not use operation logs). Unison takes into account the semantics of the file system when trying to merge two replicas. Non-conflicting updates are automatically propagated, but nothing is done with conflicting updates. Thus, after reconciliation replicas may hold different states.

2.2.6 CVS

The Concurrent Versions System [9] is a version control system that lets users edit a group of files collaboratively and retrieve old versions on demand. A central site stores the repository that contains authoritative copies of the files and the associated changes. Users create private copies (replicas) of the files and modify them concurrently. After that, users commit private copies to the repository. CVS automatically merges changes of distinct users on the same file if there is no overlap. Otherwise, user must resolve conflicts manually.

2.2.7 Harmony

Harmony [35] is a generic framework for reconciling disconnected updates to heterogeneous, replicated XML data. For instance, Harmony is used to reconcile the bookmarks of multiple web browsers (Mozilla, Safari, OmniWeb, Internet Explorer, and Camino). This application allows bookmarks and bookmark folders to be added, deleted, edited, and reorganized by different users on disconnected machines. Similar to Unison, Harmony takes only replica states and it does not resolve update conflicts.

2.2.8 Bayou

Bayou [34] is a research mobile database system that lets a user replicate a database on a mobile computer, modify it while disconnected, and synchronize with any other replica of the database that the user happens to find. In Bayou, each operation has attached a dependency check and a merge procedure. The dependency check is run to verify if the operation conflicts with others whereas the merge procedure is executed to repair the replica state in case of conflict. In Bayou, a single primary site decides which operations should be committed or aborted and notifies other sites about the sequence in which operations must be executed. Anyway, Bayou remains different

from single-master systems as it allows any site to submit operations and propagate them, letting users to quickly see the operations effects. In single-master systems, only the master can submit updates.

2.2.9 OceanStore

OceanStore [28] is a peer-to-peer distributed storage that targets groupware applications, as Grid4All does. It provides an update model based on conflict resolution, derived from that of Bayou. However the conflict resolution uses centralization points called primary tiers, which may render it unsuitable for large scale cooperative applications.

2.2.10 IceCube

Icecube [38, 37] is a general-purpose reconciliation system that exploits the application semantics to resolve conflicting updates. In IceCube, update operations are called actions and are stored in logs. IceCube captures the application semantics by means of constraints between actions. It treats reconciliation as an optimization problem where the goal is to find the largest set of actions that honor the stated constraints. Reconciliation is performed by a master site, which propagates the outcome to other sites.

2.2.11 Distributed log-based reconciliation

Chong and Hamadi [11] propose distributed algorithms for log-based reconciliation constructed upon the formal framework introduced in IceCube. Actions and constraints are partitioned in a set of nodes that locally compute the largest set of non conflicting actions, and then combine these local solutions into a global consistent distributed solution. This approach requires an ordering between nodes that share constraints.

So far, we have seen replication solutions in the context of distributed systems. However, P2P systems have specificities that have lead to a panel of adhoc techniques detailed in the next section.

2.3 Replication in P2P systems

P2P systems allow decentralized data sharing, distributing data storage across all peers of a P2P network. Since these peers can join and leave the system at any time, the shared data may become unavailable. To cope with this problem, P2P systems replicate data over the P2P network.

In this section, we present the main existing P2P systems from the perspective of data management and we discuss the corresponding data replication solutions. Throughout this section we assume that the reader has a good knowledge of P2P systems. For a survey of P2P systems please refer to Grid4All Deliverable D1.1.

2.3.1 Napster

Napster [1] is a P2P system supported by a super-peer network which relies on central servers to mediate node interactions. Any peer that shares files connects to a super-peer and publishes the files it holds. The super-peer, in turn, keeps connection information (e.g., IP address, connection bandwidth) and a list of files provided by each peer.

In order to retrieve a file from the overall P2P network, a peer sends a request to the super-peer, which searches for matches in its index and returns a list of peers that hold the desired file. The peer that has submitted the query then opens direct connections with one or more peers belonging to the super-peer reply and downloads the desired file.

Napster relies on replication for improving files availability and enhancing performance, but it does not implement a particular replication solution. Indeed, replication occurs naturally as nodes request and copy files from one to another. This is referred as passive replication. Napster is simple to implement and efficient for locating files, but it has two main limitations. First, it stores read-only data (e.g., music files). Second, super-peers constitute single points of failure and are vulnerable to malicious attack.

2.3.2 Gnutella

Gnutella [24] is a P2P file sharing system built on top of an IP network service. Its overlay network is unstructured. In order to obtain a shared file, the node that requests the file (the requestor) must perform three tasks: join the Gnutella network, search the desired file, and download it.

To join the Gnutella network, the requestor connects to a set of nodes already joined (a bootstrap list is available in databases such as gnutellahosts.com's one) and sends them a request. Each of these nodes receiving this request send back a message containing its IP and port as well as the number and size of its shared files; and in addition, it propagates the announcement request to its neighbors.

Once joined, the requestor can search the desired file. The searching mechanism starts with a query message q sent by the requestor to its neighbors and distributed throughout the network by flooding . Replies to q are routed back along the opposite path through which q arrived. A reply of a host that can satisfy q is called query hit and contains the IP, port, and speed of the host. When the requestor receives a query hit message, it directly connects to the node that holds the desired file and performs the download. In order to improve efficiency and preserve network bandwidth, duplicated messages are detected and dropped. And in addition, the message spread is limited to a maximum number of hops.

As Napster, Gnutella implements passive replication, i.e., a file is only replicated at nodes requesting the file. To improve locality of data, as well as availability and performance, active replication methods were proposed in which files may be proactively replicated at arbitrary nodes. However, Gnutella keeps on a major limitation, namely it deals with read-only files.

2.3.3 Chord

Chord [44] is a P2P routing and location system on top of a DHT overlay network. Chord uses consistent hashing [25] for mapping data keys to nodes responsible for them.

The consistent hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible. A *node identifier* is chosen by hashing the node's IP address, while a *key identifier* is produced by hashing a data key. All node identifiers are ordered in a circle modulo 2^m. A data key k is assigned to the first node whose identifier is equal to or follows k in the identifier space. The use of consistent hashing tends to balance load as each node receives roughly the same number of keys.

Chord does not implement data replication; it delegates this responsibility to the application. However, it proposes that the application implements replication by storing the object under several keys derived from the data's application level identifier. Knezevic et al. [27] realizes this purpose assuring that in case of concurrent updates on the same replicated object only one peer

completes the operation. In addition, missing replicas are proactively recreated within refreshment rounds. This approach gives probabilistic guarantees on accessing correct data at any point in time. Akbarinia et al. [3] use multiple hash functions to produce several key identifiers from a single key. They allow updating replicas of the same object in parallel and rely on timestamps to automatically resolve conflicts. This approach provides probabilistic guarantees of consistency among replicas; however, conflicting updates might cause lost updates.

In [19] the authors introduce propose a lighweight replication service on top of Chord that aims at reducing the nodes load. The authors propose a generic algorithm LAR and evaluates its performances while it runs on top of Chors. LAR takes a minimalist approach to replication. Servers periodically compare their load to local maximum and desired loads. High load causes a server to attempt creation of a new replica usually the sender of the last message. Since servers append load information to messages that they originate, .downstream. servers have recent information on which to base replication decisions. Information about new replicas is then spread on subsequent messages that contain requests for the same data item.

Replication on top of Chord was recently addressed in [12]. The authors propose a simple a efficient replication on top of Chord.

A major limitation of Chord is that the user cannot control data placement.

2.3.4 CAN

CAN (Content Addressable Network) [40] relies on a structured P2P network that resembles a hash table. It uses a virtual d-dimensional Cartesian coordinate space to store and retrieve $(key, value)$ pairs. This coordinate space is completely logical as it is not related to any physical coordinate system. At any point in time, the entire coordinate space is dynamically partitioned among all nodes in the system, so that each node owns a distinct zone that represents a segment of the entire space.

From the replication perspective , CAN assumes immutable (read-only) content and similarly to Chord, the main limitation of CAN is that the user cannot control data placement. The two approaches proposed in CAN to ensure replication are the following:

- The first one is to use m hash functions to map a single key onto m points in the coordinate space, and, accordingly, replicate a single $(key, value)$ pair at m distinct nodes in the network (similarly to Chord's solution).
- The second approach represents an optimization over the basic design of CAN that consists of node n proactively pushing out popular keys towards its neighbors when n finds it is being overloaded by requests for these keys. In this approach, replicated keys should have an associated time-to-live field to automatically undo the effect of replication at the end of the overloaded period.

CAN infrastructure was recently used in publish/subscribe [21] systems in order to store in a persistant way the subscribtions keys. The CAN grid is devised in two parts following the main diagonal. All data stored above the main diagonal is replicated in the zones situated under the main diagonal.

2.3.5 Pastry

Pastry [41] is a P2P infrastructure intended for supporting a variety of P2P applications like global file sharing, file storage, group communication, and naming systems. Pastry is built on top of a structured overlay network.

Each node in the Pastry network has a 128 bits long identifier. Node identifiers are ordered in a circle like Chord identifiers. Data placement in Pastry is also similar to Chord, i.e., an object identified by key is stored at the node whose id is closest to key.

Contrasting with Chord, Pastry takes latency into account to establish node's neighborhoods. For routing a message looking for a certain key k, each node forwards this message to its neighbor whose id is the most similar to k. In addition, the application is notified at each Pastry node along the message route, and may perform application-specific computations related to the message.

Pastry does not implement object replication directly, but it provides functionalities that enable an application on top of Pastry to easily take advantage of replicas. First, Pastry can route a message that looks for key to the k nodes whose ids are closest to key. As a result, a file storage application, for instance, can assign a key to a file (e.g., using a hash function on file's name and owner) and store replicas of this file on the k Pastry nodes with ids closest to key. Second, Pastry's notification mechanisms allow keeping such replicas available despite node failures and node arrivals, using only local coordination among adjacent nodes.

2.3.6 Freenet

Freenet [14] is a distributed information storage system focused on privacy and security issues. In FreeNet users contribute to the network by giving bandwidth and a portion of their hard drive (called the ”data store”) for storing files. Unlike other peer-to-peer file sharing networks, Freenet does not let the user control what is stored in the data store. Instead, files are kept or deleted depending on how popular they are, with the least popular being discarded to make way for newer or more popular content. Files in the data stor e are encrypted to reduce the likelihood of prosecution by persons wishing to censor Freenet content. Concerning the underlying P2P network, Freenet is often qualified as loosely structured network since the policies it employs to determine the network topology and data placement are not deterministic.

To add a new file, a user sends an insert message to the system, which contains the file and its assigned location-independent globally unique identifier (GUID). The file is then stored in some set of nodes. During the file's lifetime, it might migrate to or be replicated on other nodes. To retrieve the file, a user sends out a request message containing the GUID key. When the request reaches one of the nodes where the file is stored, that node passes the data back to the request's originator.

Every node in Freenet maintains a routing table that lists the addresses of other nodes and the GUID keys it thinks they hold. When a node receives a query, if it holds the requested file, it returns this file with a tag identifying itself as the data holder. Otherwise, the node forwards the request to the node in its table with the closest key to the one requested, and so forth. If the request is successful, each node in the chain passes the file back upstream and creates a new entry in its routing table associating the data holder with the requested key. Depending on its distance from the holder, each node might also cache a copy locally. An insert message follows the same path that a request for the same key would take, sets the routing table entries in the same way, and stores the file in the same nodes. Thus, new files are placed where queries would look for them.

Data replication occurs as a side effect of search and insert operations. Searches replicate data along the query paths (upstream). In the case of an update (which can only be done by the data's owner) the update is routed downstream based on keys similarities. Since the routing is heuristic and the network may change without notifying peers updates may be lost and consistency is not guaranteed.

2.3.7 Past

PAST [42] is a P2P file storage system that relies on Pastry to provide strong persistency and high availability of immutable (read-only) files on the Internet. The PAST system offers the following operations: insert, lookup, and reclaim.

The insert operation stores a file in k distinct nodes within the PAST network. The lookup operation reliably retrieves a copy of the desired file if it exists in PAST and if at least one of the k nodes that store the file is reachable. Then the file is normally retrieved from a live node "near" (in terms of latency) the PAST node issuing the

The reclaim operation reclaims the storage occupied by the k copies of a file. Once the operation completes, PAST no longer guarantees the success of lookup operations. Reclaim is different from delete because the file may remain available for a while.

Replica management in PAST is based on Pastry's functionalities.

2.3.8 DKS

DKS is a familly of infrastructures for P2P systems. Each search in DKS is resolved in $log_k(N)$ steps. To ensure this, each node is present at $log_k(N)$ levels in the hierarchical overlay DKS. At each level a node has a view. Each view is composed of k equal parts disjoint (intervals) of the identifier space. For each interval in a node n view, DKS maintains a contact point that the node will contact whenever it wishes to reach some keys in the respective view.

On top of DKS was proposed a replication service based on the symmetry principle. The main idea behind symmetric replication is that each identifier in the system is associated with f other identifiers. If identifier i is associated with identifier r, then any item with identifier i is stored at the peers responsible for identifiers i, and r. Similarly, any item with identifier r is also stored at the peers responsible for the identifiers i, and r. The identifier space is partitioned into $\frac{N}{f}$ equivalence classes such that identifiers in an equivalence class are all associated with each other. To replicate items in this scheme, the responsible peer of identifier i stores every item with an identifier associated with i. This implies that to find an item with identifier i, a request can be made for any of the identifiers associated with i.

3 Requirement analysis

Requirements on the semantic store stem from both the limitations of existing OR systems and the large-scale, dynamic, P2P nature of the Grid4All environment. We divide the requirements on the semantic store into *functional* and *technical* requirements. The former relate to what end users expect from the semantic store, whereas the latter address architectural, engineering and ressource usage issues. We further distinguish requirements regarding the semantic store itself, and those regarding the integration of the semantic store with the underlying VOFS. Next paragraphs address each of these aspects.

3.1 Functional requirements

3.1.1 Semantic store

Users connected at different locations and/or times should be able to create, view, and update shared documents. When collaboratively editing a document, users should be able to make progress despite conflicts that may arise.

In order to be relevant, conflict detection and resolution should take semantics into account. All possible levels of semantics should be adressed: data semantics, application semantics and user intents.

Users should be able to define their own view of a shared document. For instance, they should see each other updates in real time in the case of shared whiteboard. On the other hand, they should be able to select a personalized view where other updates are ignored when editing a text document, in order not to be distracted by group activity.

The semantic store should helps users with reconciliation when conflict occurs. It should present users with alternative solutions to the conflict, and let users choose the one they prefer. For instance, they should be able to retain the most important update, or minimize the amount of lost work.

Users should be able to define semantic constraints between two or more documents, in order to enforce consistency accross documents. This should be allowed even if these documents are processed by distinct applications.

Users should be able to access and modify a shared document regardless of their being connected or disconnected to/from other Grid4All computers. The only change they should perceive is the frequency of updates of the document's contents.

3.1.2 Integration with VOFS

The semantic store will allow users to share documents with rich semantics and that are updated by multiple writers. However, we expect Grid4All users to also share files with simple structure or that are updated by a single writer. Handling these *plain files* does not require the services of the semantic store and thus they will be stored directly in the VOFS for simplicity.

Users should be able to store, designate and manipulate semantic store documents and plain files in the same way. In particular, semantic store documents and plain files should share the same name space and the same storage ressources. As for plain files, users should be allowed to copy, move and delete a semantic store document and to change its access rights.

Just as VOFS guarantees ubiquitous access to plain files, the semantic store should not assume that users are tied to a specific computer of the system. Instead, users should be allowed to move from one computer to another and still access semantic store documents without any loss of functionality or performance.

3.2 Technical requirements

3.2.1 Semantic store

The architecture of the semantic store should be fully P2P and it should respects the autonomy of each site. In particular, the semantic store should not assume constant connectivity between sites. It should also allow distributed conflict resolution, with no specific leader site.

The semantic store should allow several applications to concurrently use its services at any one site. These applications should be able to run anaware of one another, unless they process documents that are bound by cross-document constraints. Moreover, handling bound documents should not require complex interfaces and synchronization mechanisms between applications.

The semantic store should optimize the use of the disk, CPU and network ressources required to store and process a shared document. In particular, logs should not be allowed to grow indefinitely and document snapshots should be stored incrementally. Conflict resolution and action scheduling should be performed only when necessary.

3.2.2 Integration with VOFS

Section 3.1.2 states that file commands such as `copy`, `move`, etc., should have their counterpart for semantic store documents. To avoid developping a new set of commands, semantic store documents should actually be handled through *regular* file system commands. This is particularly important considering that a Grid4All system will aggregate computers with various operating systems. This requirement implies that semantic store documents be stored under a regular file system structure.

4 Semantic store - Telex

We propose to use Telex as the semantic store layer of Grid4All. Telex enables collaborative application by providing consistency services based on the Action Constraint Formalism (ACF).

Telex defines a *document* as the basic sharing unit. The current state of a document is represented by the history of *actions* submitted by users to update the document. These actions are bound by *contraints* that express the semantics of updates by defining scheduling invariants between actions.

The ACF defines three elementary constraints: `non-commuting`, `not-after` and `enables`. These constraints can be combined to express a wide range of semantics: data semantics, such as commutativity and conflicts, application semantics, such as causal dependence or user intents, such as atomicity. A detailed description of the ACF can be found in Grid4All Deliverable D1.3.

Telex replicates the history of actions and constraints across co-operating sites. Based on locally-known actions and constraints, Telex computes *sound schedules*, i.e. sequences of actions that comply with constraints, and proposes them to the application for execution.

Telex ensures that co-operating sites eventually apply the same schedule on their local replica by periodically running a distributed committment protocol. Each site votes for the schedule of its choice, possibly specified by the local user. No site takes precedence over the others.

A user may select a particular view of a document by means of *action filters*. A filter defines which actions of the document history Telex must exclude when computing sound schedules. The user may define several filters on a document and dynamically add and remove them.

A user may take one or more *snapshots* of a document of particular interest to him. Telex takes additional snapshots in order to speed up the traversal of action history. Snapshots are also used as garbage collection points of the history of actions and constraints.

Although documents are the basic sharing unit, Telex does not assumes that documents are always independent. Instead, it allows a user or an application to define a constraint between actions of two distinct documents. In this case, Telex computes sound schedules over the actions of all of the documents bound by constraints.

The remainder of this section is organized as follows. Subsection 4.1 presents the main data structures of Telex. Subsections 4.2 to 4.4 describe the architecture of Telex and its operation. Subsection 4.5 presents the interface that Telex exposes to applications.

4.1 Data structures

4.1.1 Document

The document is the basic data sharing unit in Telex. Figure 1 shows the various components of a Telex document, as stored on persistent storage. These are:

- A multi-log. This is the main data structure, which represents the contents of the document. It contains the history of actions and constraints submitted by users. The multi-log is replicated asynchronously at each participating site.
- A set of user-defined action filters. Filters are saved as part of the persistent state of the document, in a per-user name space. Filters are not replicated on each participating site. Depending on Telex's *operating mode*, described later, filters may however be accessible from all sites.
- A set of user-defined state snasphots. Snapshots are saved as part of the persistent state of the document, in a per-user name space. Like filters, snasphots are not replicated but may be accessible from all sites depending on Telex's operating mode.

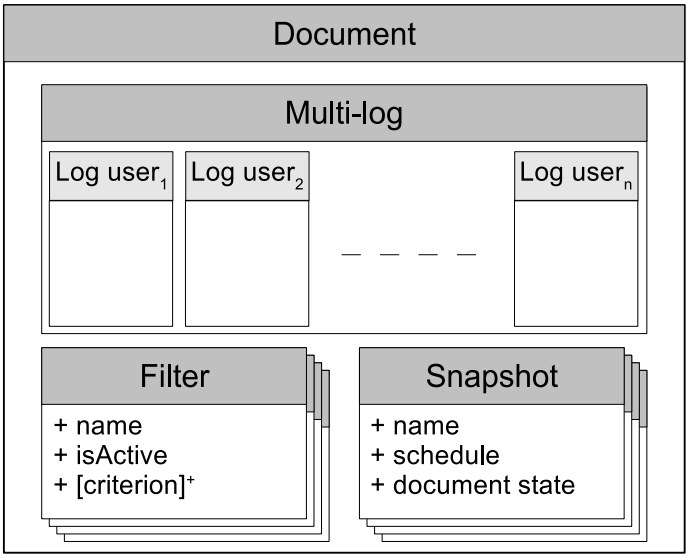


Figure 1: Components of a Telex document.

These components are described in more detail below.

4.1.2 Multi-log

The multi-log is implemented as a set of logs, one per participating user. Each log is a sequence of XML records representing either an action or a constraint.

To ease garbage-collection, Telex splits per-user logs into *chunks*, not shown on Figure 1. Thus, triming the history of actions and constraints from the head simply amounts to delete some log chunks.

4.1.3 Action

An action represents an application operation. As shown in Figure 2, it is described by several attributes. Some are known to Telex, others are opaque. Attributes known to Telex are:

- issuer: the user that submitted this action.
- timestamp: the sequence number of this action, relative to issuer.
- time: the time this action was created.
- keys: keys identify the application object(s) that this action targets; their use is described in section 4.2.1.

Opaque attributes are only used by the application. They describe the semantics of the action. These are:

- operation: the application operation that this action corresponds to.
- arguments: the list of arguments of the operation.
- other: any application-specific information.

An action belongs to only one document. It is uniquely identified by the triple `(document, issuer, timestamp)`. Telex logs an action in the log of the user who issues it. The on-disk representation of an action does not include its `issuer` attribute as it is implicit.

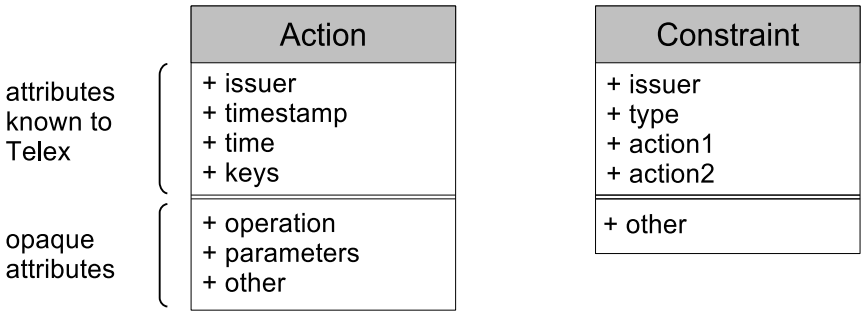


Figure 2: Action and constraint attributes.

4.1.4 Constraint

A constraint reifies a semantic relation between two actions. As shown in Figure 2, it is composed of the following attributes:

- issuer: the user that defined this constraint.
- type: the type of this constraint (`non-commuting`, `not-after`, etc.).
- action1, action2: the actions that this constraint binds.
- other: any application-specific information.

A constraint is uniquely identified by the triple `(type, action1, action2)`. Telex logs a constraint in the log of the user who issues it. The on-disk representation of a constraint does not include its `issuer` attribute as it is implicit.

Most often, a constraint binds two actions of the same document, whether issued by the same user or not. Such a constraint is called an *intra-document* constraint. However, a constraint may bind actions of two distinct documents. Such a constraint is called a *cross-document* constraint. It is then logged in *both* documents.

A constraint C references an action A by using one of the three following forms: `(timestamp)` if A is issued by the same user as C and belongs to the same document, `(issuer, timestamp)` if A belongs to the same document as C and `(docId, issuer, timestamp)` otherwise. In the latter form, `docId` is the numerical id of the document that action A belongs to. This id is an index in a *document table* that gives the full pathname of the document. This table, not shown in Figure 1, is saved as part of the persistent state of the documents in which the constraint is logged.

Figure 3 shows an example of the two types of constraint. Constraint C_1 is an intra-document constraint: it binds actions A_1 and A_2 of $\texttt{document}_1$. Constraint C_1 is issued by $\texttt{user}_1$ and thus it is logged in $\texttt{user}_1$'s log of $\texttt{document}_1$. On the other hand, constraint C_2 is a cross-document constraint: it binds action A_3 of $\texttt{document}_1$ and action A_4 of $\texttt{document}_2$. Constraint C_2 is issued by $\texttt{user}_3$ and thus it is logged in $\texttt{user}_3$'s log of both $\texttt{document}_1$ and $\texttt{document}_2$.

4.1.5 Filter

A filter defines a set of actions that a user wishes to exclude from his view of the document. The attributes of a filter are shown in Figure 1. To define a filter, a user specifies its *name* and one or more filtering *criteria*. These criteria may relate to any attribute of an action. Using generic attributes, a user may for instance filter out all of the actions issued by some user u_i. He may also filter out all of the actions that u_i submitted after a specific action or after a specific time.

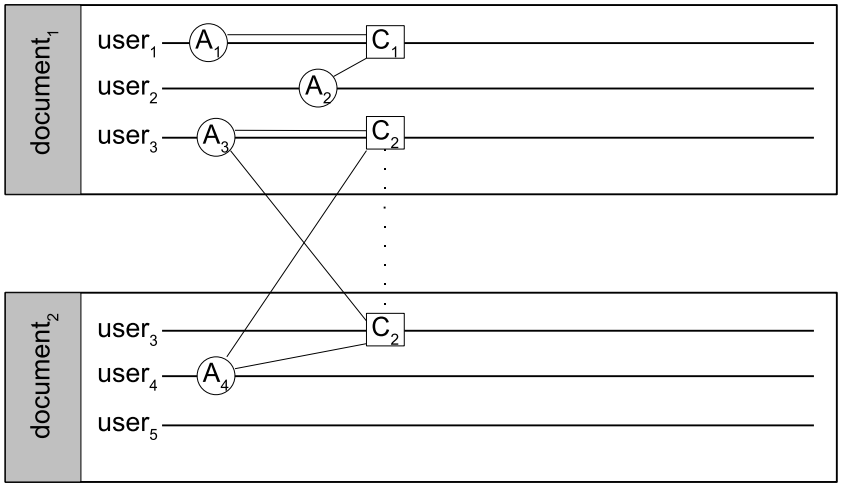


Figure 3: Intra- and cross-document constraint logging.

User may dynamically assign and remove filters to/from a document. A given filter may be assigned to one ore more documents. Telex saves the set of currently-defined filters as part the persistent state of the document. It uses a distinct name-space for each user to avoid name conflicts. When a user opens a document, Telex automatically loads and apply the filters that this user have defined.

As shown in Figure 1, user may activate or de-activate a filter. When computing sound schedules, Telex only applies the set of *active* filters. The activity status of a filter is saved as part of the persistent state of the document when the document is closed.

Note that user may define a filter that targets a specific action of a document. By activating and de-activating the filter, user may thus selectively undo and redo the corresponding action in his view of the document. (To undo an action persistently, the user must *abort* it. By convention, this is expressed by a reflexive `before` constraint on the action.)

4.1.6 Snapshot

A snapshot records the state of the document at a particular point in time. The attributes of a snapshot are shown in Figure 1. To define a snapshot, a user specifies its *name* and the *schedule* of actions whose execution yields the state being recorded. In addition, the application may provide the corresponding binary state of the document. In this case, the snapshot is said to be *materialized*.

The user may define any number of snapshots of interest to him, and later remove those that are no longer useful. Telex saves the set of currently-defined snapshots as part the persistent state of the document. It uses a distinct name-space for each user to avoid name conflicts.

When saving binary states on persistent storage, Telex optimizes disk utilization by using a content-based naming scheme, as follows. Telex first splits state into fixed-size fragments and computes the SHA-1 hash of each fragment. It then saves each fragment under the (file) name representing its hash value. File names of fragments that make up the state are stored in a file representing the snaphot. Thus, state fragments that remain unchanged from one snapshot to another are reused instead of being stored twice. Moreover, fragments are stored in a space common to all users, thus enabling fragment sharing between users. This is particularly useful when dealing with large documents, such as video files.

The size of fragments is a parameter that application may specify on a per-document basis.

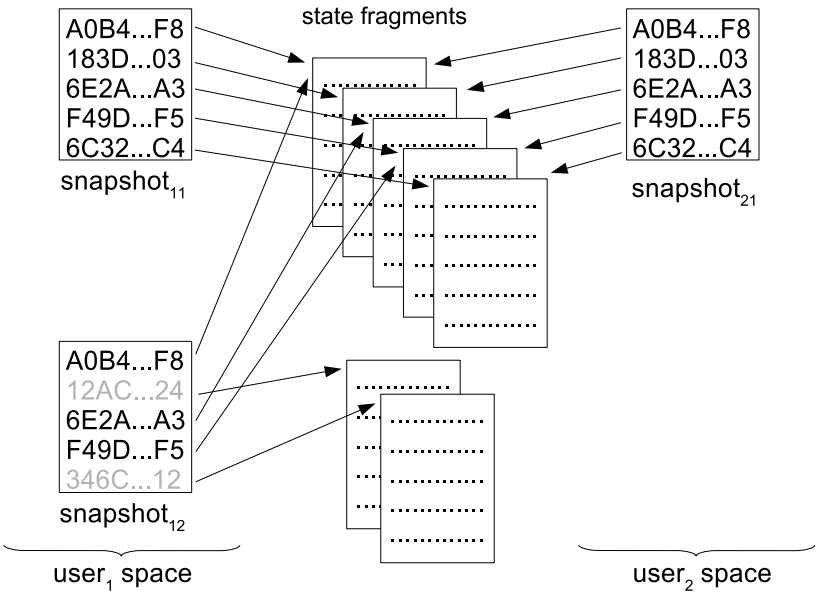


Figure 4: Snapshots and state fragments.

Moreover, based on their knowledge of structure of the document state, applications may provide a specific fragmentation scheme to Telex in order to improve fragment re-use.

Figure 4 illustrates the re-use and sharing of state fragments. It represents three materialized snapshots of the same document: $\texttt{snapshot}_{11}$ and $\texttt{snapshot}_{12}$, taken by user $\texttt{user}_1$, and $\texttt{snapshot}_{21}$ taken by $\texttt{user}_2$. Each of the corresponding binary states consists of five fragments. In this example, snapshots $\texttt{snapshot}_{12}$ and $\texttt{snapshot}_{21}$ happen to represent the same document state. Thus, although they are saved by distinct users, they share the same set of state fragments. Snapshot $\texttt{snapshot}_{12}$ was taken shortly after $\texttt{snapshot}_{11}$ and it only differs from the latter by fragments #2 and #5: fragments #1, #3 and #4 are re-used from snapshot $\texttt{snapshot}_{11}$. (Note that this figure does not represent the schedule that each snapshot file normally contains.)

4.2 Architecture and basic operation

Figure 5 shows the overall architecture of a Telex instance. One such instance runs at each site and communicates with remote instances.

On top of the figure are the applications using the services of Telex. Several such applications may run concurrently at the same site. These applications may run on behalf of distinct users, possibly belonging to distinct VO. A Telex application may be an application developped from scratch on Telex, like the shared Calendar described in Chapter I. Or it may be a adapter layer inserted beneath an existing application, like for example a database manager.

In the middle of the figure is the Telex middleware. It is composed of two main modules — the scheduler and the replica reconciler — layered on top of two auxiliary modules — the transmitter and the logger. Arrows in the figure represent invocation paths between Telex modules and to/from applications.

Each application may open one or more documents. For each open document, Telex creates one instance of each module, which maintains the execution context of the document. The only exception is when documents are bound by cross-document constraints, as described in section 4.3.1.

Each document is associated with a set of Telex instances that collaboratively edit the docu-

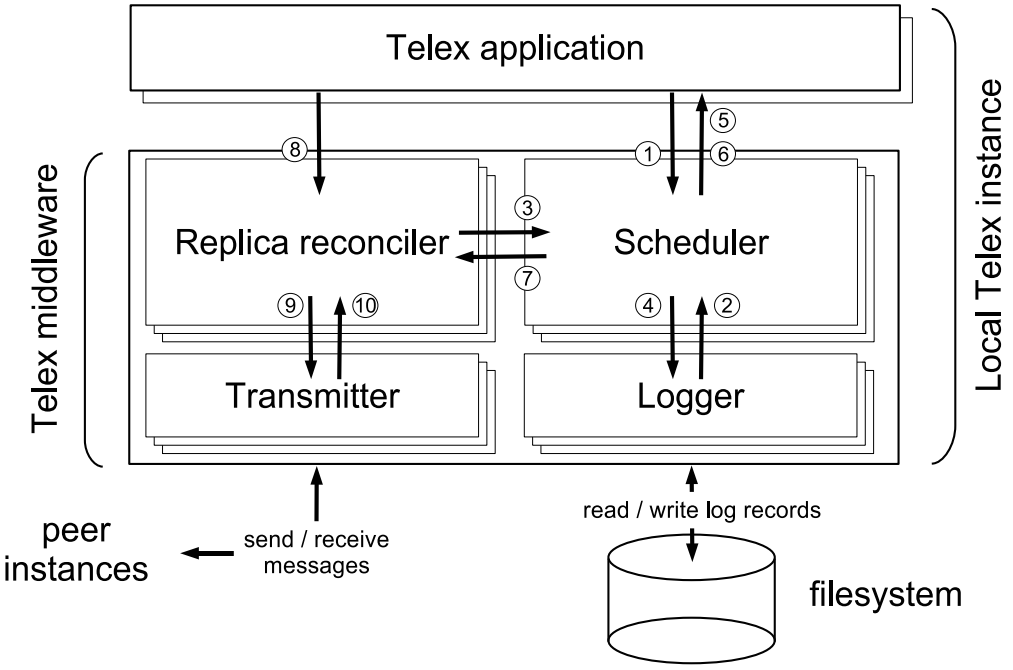


Figure 5: Telex architecture

ment, either at the same time, earlier or later. This set is specific to a document (or to a group of documents) as a user is involved in several working groups in the general case.

The operation of each module is described next. In the description, *the document* refers to the document the module has beeen instanciated for, *the application* is the application that currently edits the document and *peer instances* refers to Telex instances that collaboratively edit the document.

4.2.1 Scheduler

The role of the scheduler is twofold. First, it maintains an action-constraint graph that represents the local state of the document. Second, it periodically computes sound schedules from this graph and proposes them to the application for execution.

The action-constrain graph is the in-memory representation, at the local replica, of the document's multi-log. Actions and/or constraints are added to the graph either by:

- the application (arrow #1), when local user updates the document,
- the logger (arrow #2), when it receives an update issued by a remote user,
- the replica reconciler (arrow #3), when it commits a schedule.

The scheduler passes locally-submitted actions and constraints to the logger (arrow #4) to log them on persistent storage.

Cross-site constraint generation Two Telex instances may concurrently submit semantically-related actions. Thus, whenever a new action is addded to the graph, Telex must find out if a constraint exists against concurrent actions, i.e. actions submitted by a site distinct from the one submitting the new action. Telex does this in two steps. First, it quickly checks if any of the keys

of the new action is identical to a key of a concurrent action. If so, Telex then asks the application for constraints (arrow #5) and adds to the graph the constraints that the application returns, if any.

Action keys are thus used to improve performance of constraint checking by avoiding unnecessary upcalls to the application. Note that keys are opaque to Telex, which only tests them for identity. The application uses keys as a compact (but approximative) representation of the application objec(s) that the action uses or updates. Note that if two unrelated actions happens to have equal keys, no harm is done other than a loss of performance.

Periodic schedule generation Computing sound schedules is CPU-intensive. To amortize this cost, Telex computes schedules (arrow #6) only when one the following is true:

- The number of fragments added to the graph since the last round exceeds a given value (threshold parameter).
- The time since a new fragment was added after the last round exceeds a given value (delay parameter).
- The application explicitly requests schedule computing.

The application may set the threshold and delay parameters on a per-document basis and change them over time. Specific values direct Telex to compute sound schedule whenever a new action is added to the graph, or on application request only.

On-demand schedule generation A large number of sound schedules exist for any given action-constraint graph in the general case. It is therefore not feasible to compute all sound schedules beforehand and present them to the application. Besides, the application may be interested only in a few or even just one schedule. For these reasons, Telex generates sound schedules dynamically, upon application request (this is not shown in the figure). The application may thus iterate through the proposed schedules and stops when one or more appropriate schedules are found.

When generating a sequence of sound schedules, Telex runs a heuristic algorithm that tries to satisfy the following properties:

- Only one of each set of equivalent schedules (according to non-commuting constraints) should be handed to the application.
- In case of conflict between actions of local user and that of remote users, schedules containing actions of local user should be handed to the application first.
- Schedules should include as many of the actions of the action-constraint graph as possible.

4.2.2 Replica reconciler

Peer instances may generate different sound schedules from the same action-constraint graph. The role of the replica reconciler is to make peer instances agree on a common schedule to apply and thus achieve (eventual) mutual consistency. The agreed-upon schedule is said to be *committed*.

The replica reconciler implements a decentralized asynchronous commitment protocol based on voting. Periodically or on user request, each site proposes and votes for one or more schedules generated by the scheduler (arrow #7). Local user may specify the schedule(s) of his choice, if any (arrow #8). Votes are sent to peer instances (arrow #9) and the schedule that receives

(arrow #10) a majority or a plurality of votes is committed. The committed schedule is then materialized as a set of constraints added to the action-constraint graph (arrow #3).

An important feature of the protocol is that it is fully asynchronous. It runs in the background and each instance determines locally when a schedule has won an election. Meanwhile, the scheduler keeps proposing (tentative) sound schedules to the application.

Note also that the commitment protocol may run only on a subset of peer instances. Moreover, the voting process is automated and does not require user intervention.

The detailed protocol can be found in [46].

4.2.3 Transmitter and Logger

The transmitter provides a message-passing interface between local and peer intances. Its role is twofold. First, it determines the set of peer instances that edit the document. Second, it provides a broadcast service among peer instances (arrows #9 and #10). This service is best-effort: it does not provide any ordering or delivery guarantee. Modules that use the transmitter must implement a retransmission protocol if needed.

The logger interfaces the local Telex instance with persistent storage. Its role is to log actions and constraints submitted locally (arrow #4), while feeding the scheduler with actions and constraints submitted by remote instances (arrow #2). Not shown in the figure, the logger is also responsible for storing filters and snapshots. The logger hides to other modules the details of the implementation of the document presented in section 4.4.

Operating mode Telex may run in two modes: *stand-alone* and *above-VOFS*. In stand-alone mode, Telex does not use any external support. In above-VOFS mode, Telex leverages replication and communication services that VOFS provides, as described in sections 5.1.1 and 5.1.3. This allows users to manipulate Telex documents and plain files in the same way and to use Telex with more flexibility.

The scheduler and the replica reconciler are not aware of the current operating mode. It only affects the operation of the transmitter and the logger, as described next.

Stand-alone mode In stand-alone mode, each Telex instance is responsible for maintaining a replica of the multi-log of the document at the local site. To do so, the transmitter embeds a P2P communication library and it co-operates with the logger to implement and epidemic replication protocol based on vector clocks.

In this mode, Telex knows peer instances through static configuration files. Thus, a user has to configure these instances for each working group he belongs to. In addition, user mobility is restricted because Telex does not provide any remote file access service. Consequently, filters and snasphots are only available on the computer on which they were saved.

Figure 6 shows the interaction between the transmitter and the logger in stand-alone mode. Messages exchanged between peer instances are of two types: votes for schedules and multi-log records. The transmitter dispatches incoming messages according to their type. Vote messages are passed to the replica reconciler, while multi-log messsages are handed to the logger. The logger logs onto local storage all of the multi-log records it receives, whether local or remote. Conversely, it sends through the transmitter all of the multi-log records that the local scheduler submits.

Above-VOFS mode In above-VOFS mode, VOFS is responsible for maintaining a replica of the multi-log of the document at each site that accesses it. VOFS also handles communication between peer Telex instances: Telex does not use any internal communication library.

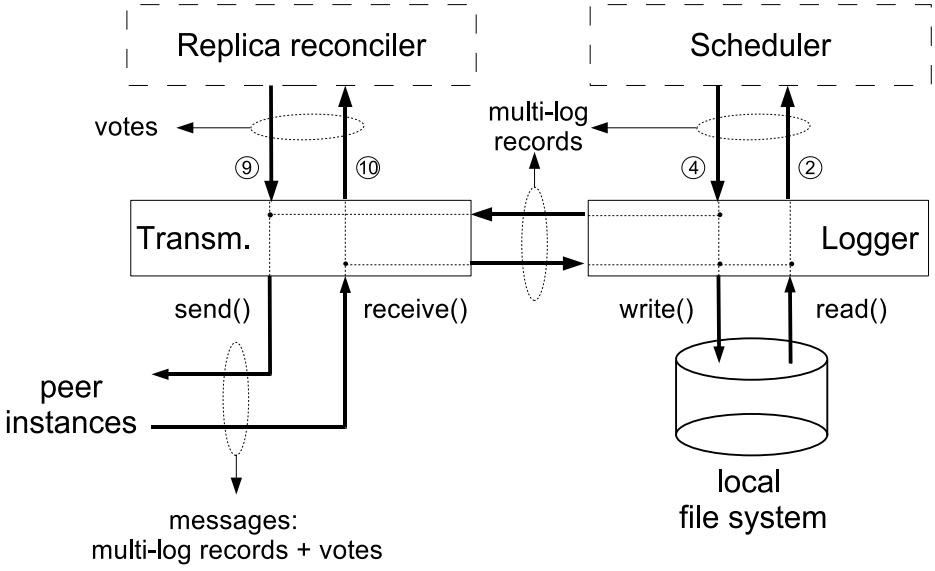


Figure 6: Stand-alone operation

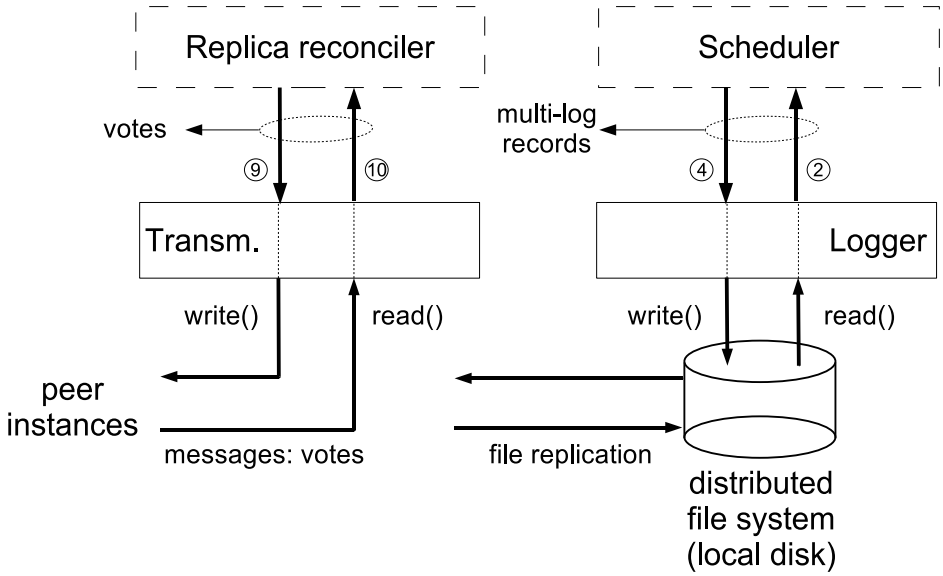


Figure 7: Above-VOFS operation

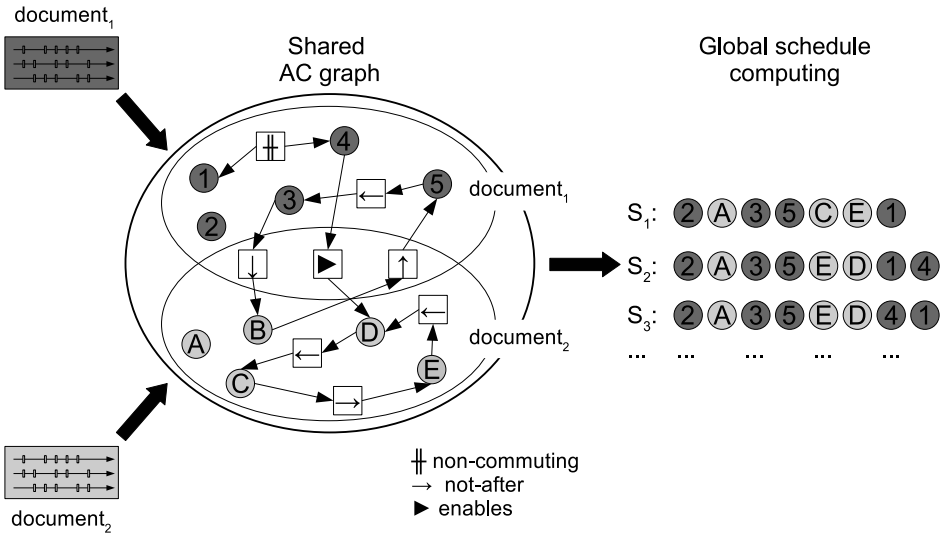


Figure 8: Bound document processing, step 1.

In this mode, VOFS dynamically determines peer instances by keeping track of the computers that open the document. VOFS also provides transparent access to remote files as a basic service. Therefore, a user may access the filters and the snasphots that he has defined from any computer. Thus, using Telex in VOFS mode eliminates the need of configuring peer instances and enables user mobility.

Figure 7 shows the operation of the transmitter and the logger in Grid4All mode. Messages exchanged between peer instances are only vote messages. Thus, the transmitter and the logger do not interact, unlike in stand-alone mode. The transmitter passes all of the (vote) messages it receives to the replica reconciler. Conversely, the logger does not send any multi-log record through the transmitter since local updates are propagated to peer instances by the replication service of VOFS.

4.3 Advanced features

Previous section described the basic operation of Telex. The following paragraphs present advanced features.

4.3.1 Bound documents

Bound documents are documents that are bound by one or more cross-document constraints. This implies that the actions of a bound document can not be scheduled independently from those of the documents it is bound to. In addition, bound documents may be handled by distinct applications. For these reasons, they require special processing.

Telex processes bound documents in two steps, as follows. First, Telex merges the actions and the constraints of the bound documents into a single *shared* action-constraint graph in order to compute *global* schedules over all actions and constraints. This first processing step is shown in Figure 8. In this example, documents `document`$_1$ and `document`$_2$ are bound by three cross-document constraints. These bind actions 3 and `B`, 4 and `D`, 5 and `B`, respectively. Twelve global schedules[1] can be drawn from this graph, of which three are shown in the figure.

[1]There are two independant `not-after` cycles of three actions each and two `not-commuting` actions in every schedule containing action `D`, which yields: $1(D) \times 3 \times 2 + 2(D) \times 3 = 12$

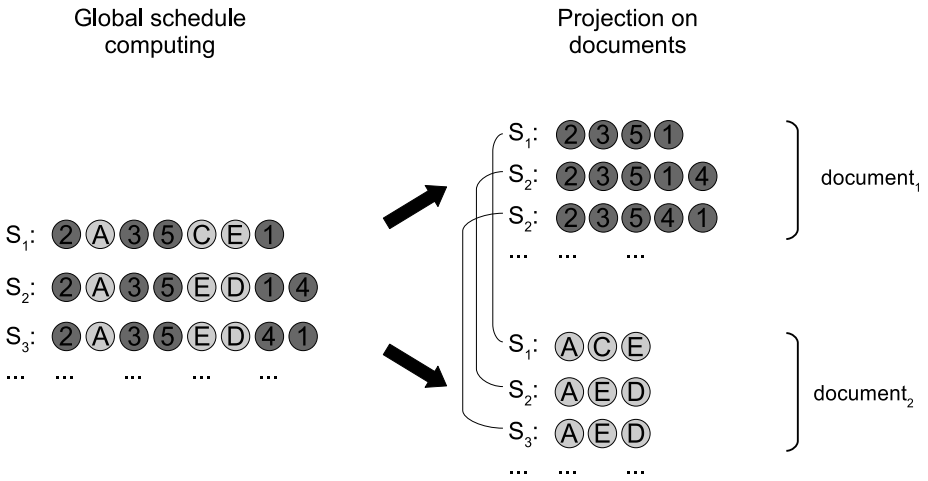


Figure 9: Bound document processing, step 2.

Each global schedule generally contains actions from all bound documents. However, the basic execution model of Telex assumes that an application applies to a *specific* document a sequence of actions *relating* to this document. Thus, in order to execute a global schedule, Telex must first project the schedule on each document, and then pass each projected schedule to the relevant application for execution. The projection operation simply consists in retaining only those actions that belong to the target document while preserving their order. This second processing step is shown in Figure 9. Global schedules S_1, S_2 and S_3 from previous example are each projected on $\texttt{document}_1$ and $\texttt{document}_2$, yielding a total of six projected schedules.

As with a non-bound document, global schedules over bound documents must be displayed to the user, so that he can specify the schedule he prefers. As noted above, the applications that handle the bound documents are likely to be distinct. From an engineering perspective, Telex applications should remain as independent as possible, if not unaware of each other. Indeed, an application cannot implement a specific interface to *every* other application in order to process bound documents and to display global schedules.

To solve this problem, Telex implements a simple solution based on naming. In this solution, applications do not interface with each other. They only interact with Telex through the basic execution model, as for non-bound documents. As shown in Figure 8, Telex identifies each global schedule with a unique identifier — S_1, S_2 and S_3 in the example. It then assigns this identifier to each projected schedule that derives from the global schedule, as shown in Figure 9. Finally, applications indicate the identifier of the schedule corresponding to the state of the document they are displaying. In this way, the user can identify matching schedules on each document. In turn, he may specify the one he prefers through one application or the other. Telex will retrieve the corresponding global schedule thanks to the unique identifier.

The identification of schedules can be achieved as shown in Figure 10. The figure represents the display of the user and the two documents of the above example. These documents are edited in a separate windows by their respective application, as if not bound. Each application displays the possible states of the document in a distinct *tab*, named after the corresponding schedule. Alternatively, applications may use a color code, info bubles or revision marks to identify all possible state within a single tab or window.

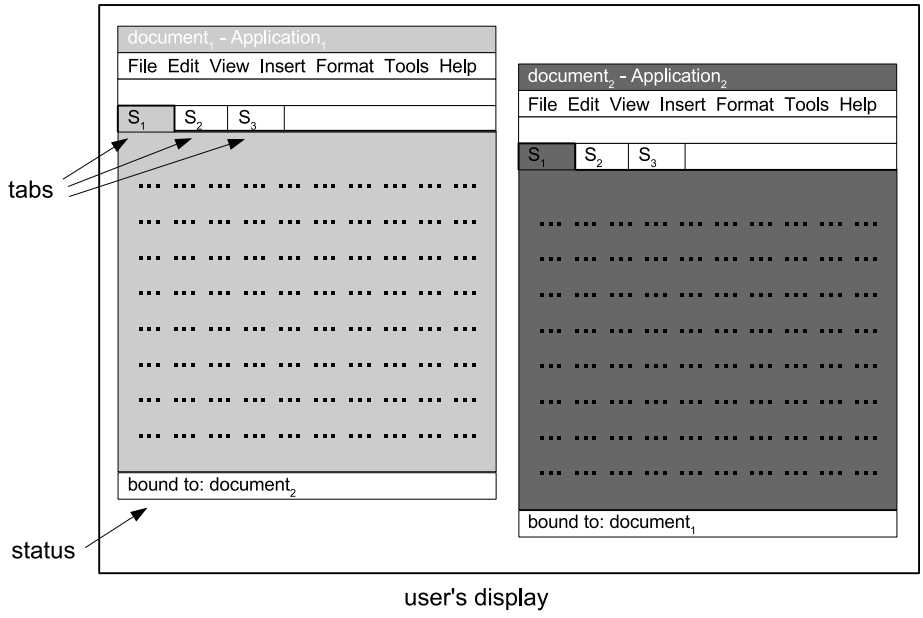


Figure 10: Displaying global schedules on bound documents.

4.3.2 Functional extensions

The scheduler and the replica reconciler modules described above implement general-purpose algorithms that suit most needs. However, applications with specific requirements may replace any of these modules with its own to override the default behaviour of Telex. Application-specific modules must implement the same interface as their Telex counterpart and they must interact with other modules as described in Figure 5.

As an example, the Calendar application described in Chapter I provides Telex with its own replica reconciler. Indeed, no distributed commitment protocol is needed in this case since each user must be free to arrange his own agenda.

Some other applications may choose to replace the default scheduler with one that gives them full access to the action-constraint graph. In this way, these applications can compute sound schedules by taking the semantics of actions into account, in addition to constraints.

4.4 Document implementation

Figure 11 depicts the actual layout of a Telex document on a file system. A document is implemented as a *directory* that contains all of the items listed in pararagraph 4.1.1. This directory is referred to as the *root directory* of the document.

Thanks to this implementation, a user may manipulate a Telex document as a single entity and (almost) as simply as a plain file. Indeed, commands for copying, moving, deleting and setting the access rights of files apply to directories with only minor changes in parameters on most operating systems.

A user designates and manipulates a Telex document by specifying the pathname of its root directory. The internal structure structure of the document remains opaque to the user and is only known to Telex.

Next sections describe the internal structure of a Telex document and how access control is achieved.

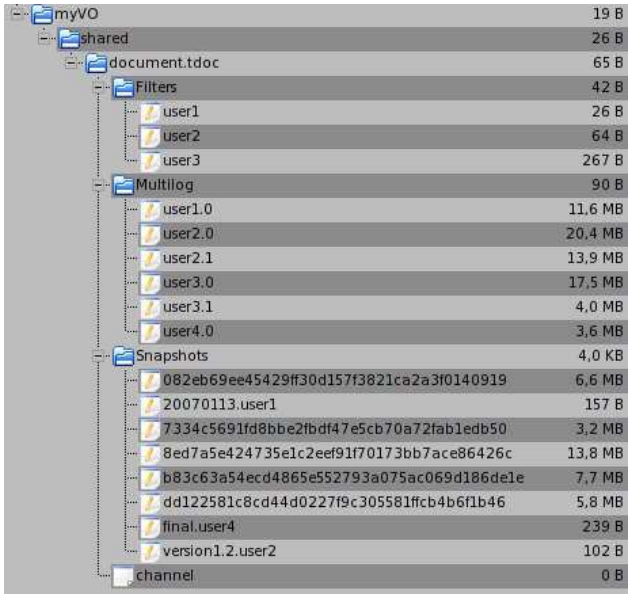


Figure 11: Internal structure of a Telex document

4.4.1 Internal structure

As Figure 11 shows, the root directory is subdivised into sub-directories containing the multi-log, filters and snapshots, respectively.

The `Filters` sub-directory contains user-defined filters. All of the filters defined by a user are stored in a single file, named after this user.

The `Multilog` sub-directory contains log chunks. Each chunk is named after the user who writes it and is suffixed with a sequence number relative to user. It is important to note that each log chunk is thus a single-writer append-only file.

The `Snapshots` sub-directory contains user-defined snapshots and the corresponding state fragments. Each snapshot is stored under the name assigned by the user, suffixed with the name of the user. State fragments are stored under the value representing the hash of their contents.

When Telex runs in above-VOFS mode, a document also contains a file named `channel`. This file materializes the communication channel used by peer instance of the document, as explained in section 5.1.3.

4.4.2 Access control

Telex defines read and write permissions on a document. Telex only relies on the mechanisms provided by the underlying file system to enforce access control. Applications may further restrict access rights with additional mechanisms of their own, but this is out of the scope of this document.

4.5 Application API

Figure 12 gives an overview of the interface between Telex and the application. The interface comprises downcalls (from application to Telex) and upcalls (from Telex to application) corresponding to arrows #1, #5, #6 and #8 of Figure 5. Note that this figure does not mention miscellaneous calls, like calls that open or close a document, or get the status of a document.

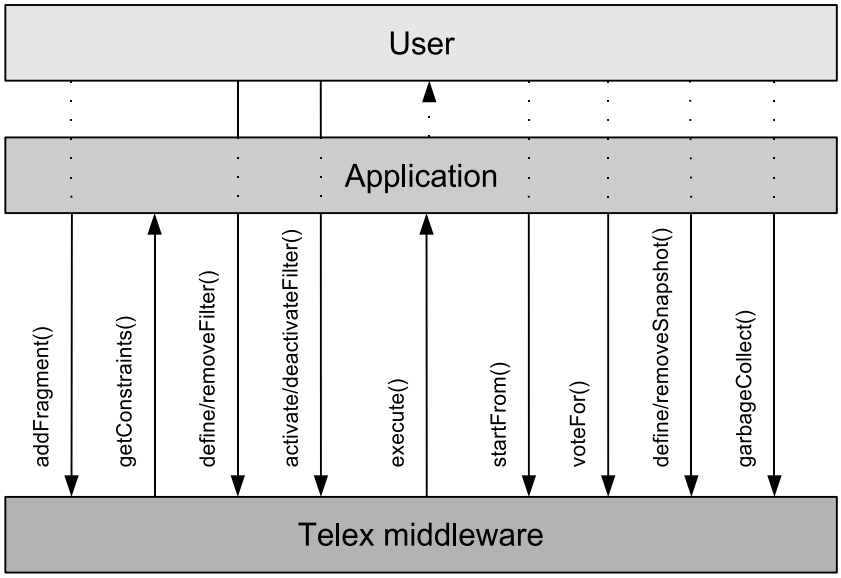


Figure 12: Telex Application API.

Figure 12 also shows the interaction between application and user. Solid lines denote calls that are triggered only by the user. Dashed lines denote calls that may be triggered either by the application or by the user.

The detailled description of the interface can be found in Appendix I.

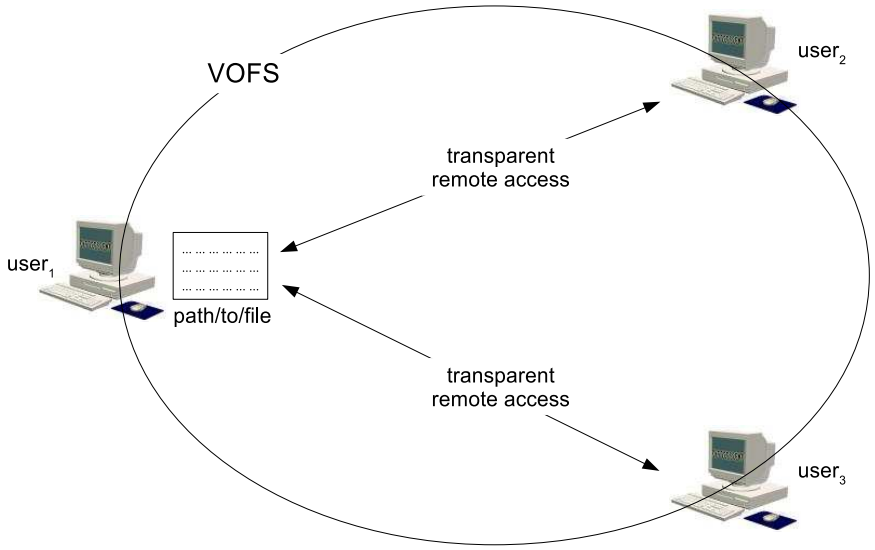


Figure 13: Sharing data through VOFS.

5 Interfacing Telex with VOFS

Telex interfaces with VOFS in order to store and retrieve Telex documents. Figure 13 illustrates the basic principle of data sharing in Grid4All. In this example, three users share their files and disk ressources through the VOFS. User $\texttt{user}_1$ has created a plain file under the `path/to/file` pathname in the VOFS. The file happens to reside on the computer of $\texttt{user}_1$ because `path/to` refers to a directory on his local disk. Users $\texttt{user}_2$ and $\texttt{user}_3$ may refer to `file` through its full pathname. If $\texttt{user}_1$ has granted them access, they may read and write `file` through the VOFS as if it were local. This only requires the basic services that a regular distributed file system provides, namely transparent file location, transparent file access and file access control.

This scheme also works for a Telex document, even though its structure is more complex than that of a plain file. Indeed, VOFS allow $\texttt{user}_2$ and $\texttt{user}_3$ to write their log and read other users' log remotly on the computer of $\texttt{user}_1$. Read and write operations do not raise any consistency issue since each log is a single-writer file. Thus, no additional service is required to handle Telex documents because of their specific structure and semantics.

However, the basic services mentioned above are not sufficient to handle the dynamic, P2P environment that Grid4All targets. This holds for plain files as well as for Telex documents. Indeed, VOFS must allow for disconnections and preserve the automony of users and sites. To this end, VOFS provides enhanced services regarding file replication, event notification and communication. These are are described in detail in Deliverable 3.2 "Interface specification and initial running prototype of the Virtual Block Store - DFS+VBS Architecture and Prototype".

Next sections describe how Telex leverages these enhanced services and present the corresponding API.

5.1 Enhanced services

5.1.1 File replication

A process accessing a (remote) file may request VOFS to create a persistent replica of the file on the local disk. If at most one process writes to the file at a time, VOFS guarantees that the file and the local copy are eventually consistent despite possible network failures. When network

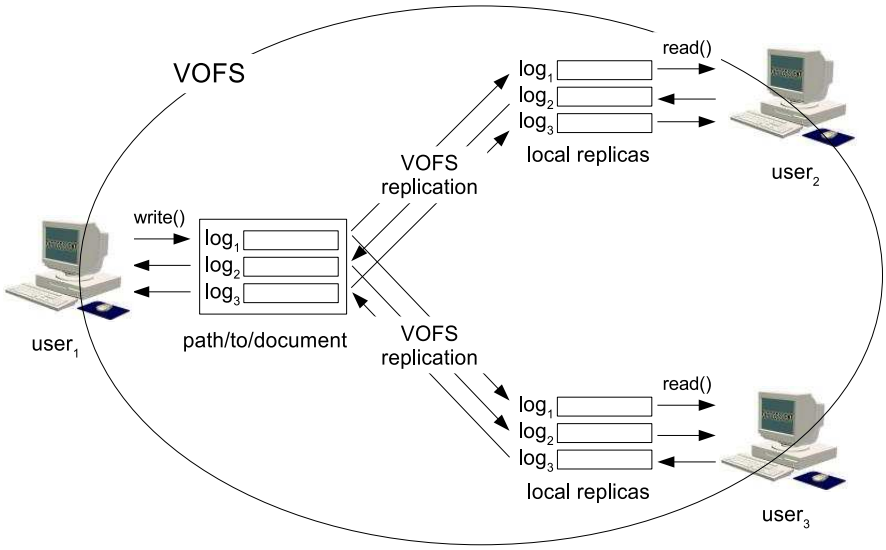


Figure 14: Replicating the multi-log through VOFS.

links are up, VOFS propagates updates of the remote file as soon as possible. When a link is down, VOFS buffers the updates and propagate them when link is up again. This service may be activated on a per-file basis at open time.

Telex leverage this service to create local replica of a document's multi-log at each site that edits the document, as shown in Figure 14. Note that it can do so because the multi-log is implemented as a set of *single-writer* logs. Thanks to the local replica, a user who happens to be disconnected from its peers may still access and update the shared document. For the same reason, Telex requests that the filters and the snapshots that the local user creates be replicated locally.

5.1.2 Event notification

VOFS allows a process to be notified of various events affecting a file system entry, whether a file or a directory. The type of event may be configured dynamically on a per-entry basis. Telex requests the notification of three types of event, as described next.

Directory update notification When opening a document, Telex requests VOFS to be notified of a file being created in the `Multilog` sub-directory of the document. Such an event occurs when a user creates his log for the first time or when he starts filling in a new log chunk. In either case, Telex must read the new log in order to update the in-memory action-constraint graph. The notification service avoid polling the `Multilog` sub-directory and provides a more efficient and more responsive mechanism.

Log update notification For the same reason, Telex also requests VOFS to be notified of a log chunk being (remotely) updated. Without the help of this service, Telex would have to poll every log chunk for updates (a read operation does not block when end of file is reached). This would be unacceptable, since the number of logs may be very large.

Directory move notification Whenever a document becomes bound, Telex requests the VOFS to be notified of the document being moved. Telex needs this notification to update all the docu-

ment tables — described in section 4.1.4 — that reference the bound document. This functionality allows users to freely move or rename bound documents while retaining the corresponding cross-document constraints.

Note that if a user deletes a bound document, he also implicitely deletes the corresponding cross-document constraints. However, Telex does not need to be notified of such event: it will simply notice that the document is deleted when trying to open it.

5.1.3 Communication

VOFS implements the file replication and the event notification services on top of an internal general-purpose publish-subscribe infrastructure. To allow applications to use this infrastructure for their own needs, VOFS exposes it through a file system-like interface.

A VOFS application may use this interface to define a basic broadcast service as follows. The group that the broadcast service targets is defined as the set of processes that have a particular file opened. The events that these processes subscribe to are write operations to the file. Broadcasting a message to the group simply amounts to write the message to the file.

Telex uses this scheme to implement the broadcast service that the transmitter module — described in section 4.2.3 — provides. When creating a document, Telex creates a file named `channel` under its root directory. Each Telex instance that edits this document opens this file in order to communicate with peer instances.

5.2 API

Most operations on a Telex document are achieved through the regular POSIX file system API: reading and writing records in the multi-log, saving an retrieving snapshots or filters. However, the POSIX API does not provide support for implementing the enhanced services described above.

To circumvent this problem, VOFS defines *virtual files* that provide access to information and services specific to VOFS. Like regular files, virtual files are accessed through the POSIX API. However, they are artifacts: they do not represent any actual VOFS resource and their contents is computed on demand. For each file system entry `path/to/entry`, VOFS defines a set of virtual files named `path/to/file/@<virtual file>`. The virtual files that Telex uses are described next.

@config. This file receives several configuration commands on the entry. For each log file, Telex issues the command instructing VOFS to create a local replica of the log.

@state. This file contains the current state of the entry, in the form of several attributes. The *offline* attribute indicates that the local replica is disconnected from the main file and is thus not synchronized with it any more. For each log file, Telex checks for this particular attribute and reports it back to the application for information.

@notify. This entry is actually a directory. It contains the `@notify/ctl` control file, that a process uses to specify which events it subscribes to. These events are received in the `@notify/publications` file by performing a read operation. Note that contrary to POSIX semantics, reads on this file block the calling process until an event is received.

@publish. This file receives events to be published to the (possibly remote) processes reading the `@notify/publications` file. These events can be written to the file either by a VOFS application or by the VOFS itself.

Telex uses the @notify and @publish files to be notified of event regarding: (i) the root directory of document, (ii) its `Multi-log` subdirectory, (iii) every log file in this subdirectory and (iv) the `channel` file.

References

[1] Napster.

[2] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, June 1989.

[3] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. *Global Data Management (Chapter Design and implementation of Atlas P2P architecture)*. July 2006.

[4] P. Albitz and C. Liu. Dns and bind. *4th Ed., O'Reilly*, January 2001.

[5] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proc. 2nd Int. Conf. on Software Engineering*, pages 562–570, San Francisco, CA (USA), October 1976.

[6] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. http://research.microsoft.com/pubs/ccontrol/.

[8] A. Bhargava, K. Kothapalli, C. Riley C. Scheideler, and M. Thober. Pagoda: a dynamic overlay network for routing, data management, and multicasting. *In Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, January 2001.

[9] Per Cederqvist, Roland Pesch, et al. Version management with CVS, date unknown. http://www.cvshome.org/docs/manual.

[10] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, December 2001.

[11] Y.L. Chong and Y. Hamadi. Distributed log-based reconciliation. *In Proc. of the European Conference on Artificial Intelligence (ECAI)*, September 2006.

[12] Maria Christodoulidou and Panagiota Fatourou. Simple and efficient replication in chord. In *Proceedings of the IASTED Parallel and Distributed Computing and Systems (PDCS'06)*, November 2006.

[13] P. Chundi, D.J. Rosenkrantz, and S.S. Ravi. Deferred updates and data placement in distributed databases. *In Proc. of the Int. Conf. on Data Engineering (ICDE)*, February 1996.

[14] I. Clarke, S. Miller, T.W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, January 2002.

[15] C. Coulon, E. Pacitti, and P. Valduriez. Consistency management for partial replication in a high performance database cluster. *In Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, July 2005.

[16] D.L. Eager and K.C. Sevcik. Achieving robustness in distributed database systems. *ACM Transactions on Database Systems*, September 1983.

[17] D.K. Gifford. Weighted voting for replicated data. *In Proc. of the ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*, December 1979.

[18] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA, December 1992. Tech. Report no. UCSC-CRL-92-52, ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-92-52.ps.Z.

[19] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. In *The 24th International Conference on Distributed Computing Systems*, March 2004.

[20] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. Dangers of replication and a solution. In *Int. Conf. on Management of Data*, pages 173–182, Montréal, Canada, June 1996.

[21] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[22] S. Jajodia and D. Mutchler. Dynamic voting. *In Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, May 1987.

[23] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, September 2003.

[24] M. Jovanovic, F. Annexstein, and K. Berman. Scalability issues in large peer-to-peer networks: a case study of gnutella. *Technical report, ECECS Department, University of Cincinnati*, January 2001.

[25] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. *In Proc. of the ACM Symp. on Theory of Computing*, May 1997.

[26] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, September 2000.

[27] P. Knezevic, A. Wombacher, and T. Risse. Enabling high data availability in a dht. *In Proc. of the Int. Workshop on Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE'05)*, August 2005.

[28] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, November 2000.

[29] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Consistent data replication: is it feasible in wans? *In Proc. of the European Conf. on Parallel Computing (Euro-Par)*, September 2005.

[30] Vidal Martins, Esther Pacitti, and Patrick Valduriez. Survey of data replication in P2P systems. Research Report 6083, Institut National de la Recherche en Informatique et Automatique (INRIA), December 2006. https://hal.inria.fr/inria-00122282.

[31] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 2000.

[32] E. Pacitti, E. Simon, and R.N. Melo. Improving data freshness in lazy master schemes. *In Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, May 1998.

[33] D.S. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *In IEEE Transactions on Software Engineering*, May 1983.

[34] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS. http://www.parc.xerox.com/csl/projects/bayou/.

[35] B.C. Pierce, A. Schmitt, and M.B. Greenwald. Bringing harmony to optimism: an experiment in synchronizing heterogeneous tree-structured data. *Technical report MS-CIS-03-42, Department of Computer and Information Science, University of Pennsylvania*, February 2004.

[36] B.C. Pierce and J. Vouillon. What's in unison? a formal specification and reference implementation of a file synchronizer. *Technical report MS-CIS-03-36, Department of Computer and Information Science, University of Pennsylvania*, February 2004.

[37] Nuno Preguiça, Marc Shapiro, and J. Legatheaux Martins. SqlIceCube: Automatic semantics-based reconciliation for mobile databases. Technical Report TR-02-2003 DI-FCT-UNL, Universidade Nova de Lisboa, Dep. Informática, FCT, 2003. http://asc.di.fct.unl.pt/~nmp/papers/sqlice3-rep.pdf.

[38] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth Int. Conf. on Coop. Info. Sys. (CoopIS)*, volume 2888 of *Lecture Notes in Comp. Sc.*, pages 38–55, Catania, Sicily, Italy, November 2003. Springer-Verlag.

[39] Norman Ramsey and Előd Csirmaz. An algebraic approach to file synchronization. Technical Report TR-05-01, Harvard University Dept. of Computer Science, Cambridge MA, USA, May 2001. http://www.eecs.harvard.edu/~nr/pubs/sync-abstract.html.

[40] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *In Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, August 2001.

[41] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Int. Conf. on Dist. Sys. Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, December 2001. IFIP/ACM. http://www.research.microsoft.com/~antr/pastry/pubs.htm.

[42] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symp. on Op. Sys. Principles (SOSP)*, pages 188–201, October 2001.

[43] Yasushi Saito and Marc Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, March 2005. http://doi.acm.org/10.1145/1057977.1057980.

[44] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. *In Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, August 2001.

[45] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, December 1979.

[46] Sutra Pierre and Barreto Joao and Shapiro Marc. An asynchronous, decentralised commitment protocol for semantic optimistic replication. Research Report 6069, INRIA, 12 2006.

[47] T. Özsu and P. Valduriez. Principles of distributed database systems. *2nd Ed. Prentice Hall*, January 1999.

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.

Project no. 034567

Grid4All

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

Deliverable 3.1: Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities - Chapter II

Due date of deliverable: June 2007.

Actual submission date: 20 June 2007.

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

Organisation name of lead contractor for this deliverable: INRIA Regal

Revision: Submitted 2007-06-20

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
ACF	Action Constraint Formalism
Grid4All	The FP6 STREP project that aims to democratise access to the Grid, through the use of peer-to-peer technologies. The co-ordinator is France Télécom; the other partners are a SME from Spain, and research labs from Greece, France, Spain, and Sweden.
OS	Operating Sysetm
SC	Shared Calendar application.
VO	Virtual organization.
SyD	System on Device middleware.

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 the chapter II of Deliverable 3.1 ”Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities”, which comprises the following parts:

Chapter I	Semantic Store
Chapter II	Collaborative Applications
Chapter III	VO-aware File system
Appendix I	Telex application API

The following persons contributed to this chapter: Sarfraz Ashfaq INRIA-Atlas, Lamia Benmouffok INRIA-Regal, Jean-Michel Busca INRIA-Regal, Vidal Martins INRIA-Atlas, Esther Pacitti INRIA-Atlas, Marc Shapiro INRIA-Regal, Patrick Valduriez INRIA-Atlas.

1 Introduction

Grid4All motivation is to bring Grid computing technologies and its advantages to millions of users beyond the academic and industrial user communities of today's Grids. Grid4All aims to extend support for dynamically forming groups within the Internet, to support scalable groupware applications in dynamic environment.

To meet these requirements, Grid4All project will provide a Grid middleware framework. It will develop a set of collaborative applications to exercise the Grid4All middleware.

Grid4All middleware integrates a Data Storage (WP3) layer adapted to large-scale collaborative data sharing applications with application dependent semantic. Semantic Store layer is a part of Data Storage. It provides semantic oriented decentralized P2P replication middleware and consistency protocols.

Atlas P2P Architecture (APPA) is an existing prototype of Semantic Store. Telex is an other implementation of the Semantic Store described in "Deliverable 3.1: Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities - Semantic Store".

A use case of collaborative applications used either by academic, industrial and domestic users are a shared calendar application and a collaborative editor.

This chapter describes the implementation of a shared calendar and a collaborative editor XWiki over the Semantic Store. The shared calendar application is implemented over Telex as Semantic Store. XWiki uses APPA as Semantic Store.

For both applications, we provide a state of the art. We present the requirement of both applications. A detailed examination of the shared calendar and XWiki shows how to interface them with the Semantic Store layer.

2 Shared Calendar application

2.1 State of the art

In the last decade a significant amount of groupware applications have been developed, and democratized. An example is group-calendaring applications. A shared calendar (SC), also called group calendar application, aims to help people organizing their agenda in a collaborative way. In fact, SC allows people to create and manage private events as well as group meetings scheduled on a collection of online calendars.

Shared calendars are the target of many researches. Leysia Ann Palen [18, 19] examined the role, the successful adoption and the use of SCs. The Calendaring and Scheduling Consortium “CALCONNECT” [3] is focused on understanding and developing new technologies for calendaring data interchanges.

Frank Dawson and Derik Stenerson authored the *iCalendar* standard (RFC 2445) for calendar data exchange [7]. It is a textual format to describe the calendar data. iCalendar allows users to send meeting invitations and modifications to other users through emails.

iCalendar standard is implemented/supported by a large number of products, some of which are Google calendar (web based calendar), Microsoft Outlook, Apple iCal, IBM Lotus Agenda and Groupwise (desktop calendars).

These applications offer multiple features including:

- Keeps track of events and appointments, allows multiple calendar views (such as calendars for ”home”, ”work”, and ”kids”)
- Share a calendar with a group of people
- Creating and managing meetings
- Inviting people to a meeting
- Notification requesting response and sending responses on an operation on an event
- Identify conflicts and free time
- Discovery services: find calendars
- Portability/Compatibility: Web-based calendars like Google calendar supports any operating systems (OS). However, desktop calendars are generally OS dependent.

These applications support the asynchronous and disconnected work. They are based on client server architecture. Thus the central server may be a bottleneck of such systems and a single point of failure.

A user posts his calendar on the server or a shared directory like iCalShare [11] and DateDex [6]. He has to import a user calendar to add an event on it. The event is recorded in the calendar server. To invite users, the inviting user sets the list of the invitees and sends them the invitation by email. He sets the respective right of each invited user: can invite other users, see the list of the invitees etc. Invited users get the invitation in their mailbox, and answer back by email too.

A user can easily identify a conflict on his calendar[1]. Nevertheless, those calendar applications generally don’t handle conflicts.

[1] Two overlapping events, or more.

Besides meetings updates are not well managed and this can lead to a non-consistent state except for Outlook and Lotus Note). In fact, suppose that $user_1$ creates a meeting and invites $user_2$. $user_2$ accepts the invitation, thus the meeting is scheduled in his calendar. If $user_2$ deletes or changes the date of the meeting, then this change will appear in $user_2$'s calendar but not in $user_1$'s.

Moreover there is no proper commitment[2]in those calendars. However a commit may be useful, especially when people have to pay for an invitee or a room.

A research calendar prototype was implemented based on *System on device* middleware (SyD) [23]. SyD is a middleware to hide heterogeneity of devices, information and databases. The coordination between calendars is based on event-and-trigger mechanisms. Calendars are not replicated, so they may be a bottleneck of the systems and a single point of failure. When one creates a meeting with a group of invitees, invitees set of subscription links to that meeting . Thus, triggers will automatically propagate updates on a meeting through the subscription links. They add some semantics on the links that become 'negotiation links'. Negotiation links allow updates only if the underling constraints are satisfied. Those constraints can be "allow changes, only if changes succeeded on each of the subscribed entities". Negotiation links serve to handle conflicts. This is pessimistic updates. thus it can be hardly achieved in a dynamic and large-scale environment. Besides, it is not adapted for asynchronous work, doesn't allow disconnected work, and doesn't support commitment.

A group calendar application can be implemented on top of replicated and semantic rich systems like IceCube [25] and Bayou [21].[3] [24, 26] describe the deployment of a shared calendar application on top of IceCube. Similarly, [8] describe how Bayou can support a shared calendar application.

Those two approaches are semantically rich, thus they handle conflicts and concurrent updates. They support asynchronous work, and commitment. Besides, IceCube supports disconnected work, allows a user to undo/redo actions and to work in isolation.

Although IceCube and Bayou replicate data, the reconciliation still centralized. Thus the reconciliation server becomes the bottleneck of the systems and a single point of failure.

The SC application, within the Grid4All environment would benefits from many mechanisms and services such as data replication, persistency and consistency model. Grid4All middleware overcomes the communication, network management and access control issues. Besides, it provides communities awareness and Virtual Organization (VO) management services. The Semantic Store (chapter I of deliverable 3.1) is semantically rich, inspired from IceCube. It provides mechanisms to achieve consistency in peer-to-peer environment. SS supports commitment, asynchronous and disconnected work. Additionally, SS provides mechanisms for undoing and redoing operations.

2.2 Requirements

2.2.1 Description of the application

Considering the rationale stated above, the shared calendar application will be developed using the SS. Before describing the operation principle we must introduce the outline of the Semantic

[2] Make changes and decision permanent. The opposite is to rollback.

[3] For more detail on IceCube and Bayou, see the deliverable "D3.1: Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities – Chapter I: Semantic Store"

Store. The SS is based on the Action-Constraint Framework (ACF). ACF is a log-based optimistic replication approach, where “actions” are application atomic operations. “Constraints” are relations between actions. The application sets the constraints to express its’ semantics and to capture user intention.

As said before, the application provides users a way to manage heir activities. Each user has his own and independent calendar filled with private events. Additionally he can organize meeting with other collaborators. He creates a “meeting object”, share it, and notifies invitees of their invitation. For that purpose, he creates an operation (action) on their respective calendar. Thus, he has to import the invitees’ calendar.

When one receives an invitation he can accept it or decline it. If he accepts it, he can collaborate to hold the meeting: he can invite other users, and modify the meeting time, and location. For that purpose he creates operations (actions) on the corresponding “meeting object” concurrently with other invitees. Consequently conflicts may appear.

As we are in an optimistic replicated environment, those actions are “tentative” until they are “committed”. In case of conflict some of them are “aborted”.

2.2.2 User requirements

The SC provides the following functionalities:

Meeting management:

- Create an event
  - Create the “object” event.
- Publish an event
  - Share it with a group without inviting anybody.
- Import a user document
  - To invite other people to an event
  - Needs the approval of that user
- Invite a user
  - Ask a user to attend the event
  - Share the event “object” with the invited user
  - Notify him of the invitation on his agenda
  - Needs the approval of that user
- Cancel invitation
  - Ask a user to cancel his coming to a meeting
  - Needs the approval of the cancelled user
- Allocate room
  - Choose a specific room for a meeting
  - Or ask for an available room
- Set/update meeting time
  - Propose a time for a meeting

  - Needs the approval of all invitees
- Cancel event
  - Propose to cancel an event
  - Needs the approval of all invitees
- Accept /decline mechanism
  - To vote on one of the previous actions if required

Collaborator management:

- Get group list and user profiles
- Invite a user to collaborate
  - Ask a user to give someone read/write access (share) to his calendar.
- Stop collaboration with a user
  - Remove the read/write access on one's calendar for a user.
- Accept/decline an invitation for collaboration
  - Give or decline read/write access on own calendar.

We can define different policies for meeting modification like priority policies. For instance, one that only allows the creator to modify meeting information, and to add/cancel invitations.

2.2.3 General architecture of Shared Calendar Application

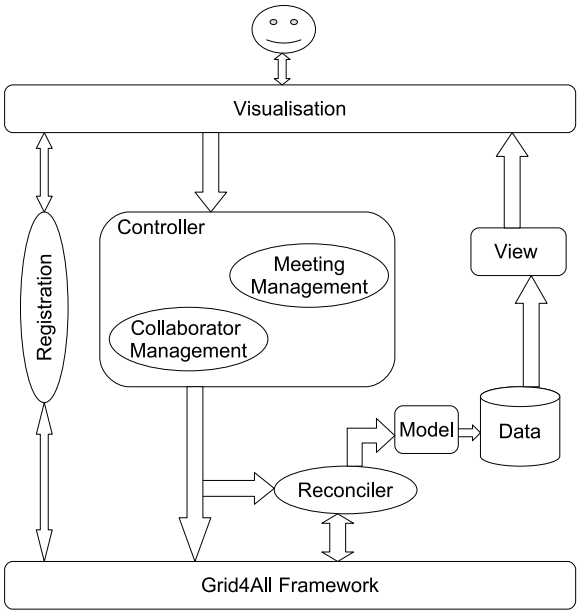


Figure 1: General architecture of Shared Calendar application.

Figure 1 shows the main components of the shared calendar application, and their interaction with the environment.

The application is based on the model-view-controller design. A **Visualisation** layer interfaces the user with the SC. A **Registration** component allows user to sign in. This registration may be hidden from the user.

The **Data** component is the materialization of the information the application needs: logs, meetings information, and collaborators' profiles

A **Controller** component captures user's modifications through the **Visualisation** layer. It generates the corresponding actions using the meeting and collaborators management functionalities.

The **Reconciler** component gets local and remote actions and compute correct schedules.

The **Model** component takes those schedules and executes them on the **Data** component.

Finally the **View** transforms the Data state into visual information.

2.2.4 Infrastructure requirements

More grid supported applications, scenarios and requirements can be found in the Grid4All deliverable "D4.1: Specification of scenarios, user requirements, and infrastructure requirements" and the deliverable "D4.2: Specification of situations derived from applications". In this section we will summarize infrastructure requirements of the shared calendar.

The application components interact with the Grid4All framework in different ways and at different levels. Those requirements and interactions with WPs are described in the following section.

Registration component requires:

- Authentication services: WP2
  - Get the user identifier
- Login service: WP3, Semantic Store
  - Login session: Inform the SS of the current user of the application.

Data component is replicated and persistent so one can keep his documents even if he works on different computers. It contains among other things users' logs. Thus, **Data** component requires:

- Replication and communication mechanism: WP1, WP3
  - Hidden by WP3
- Log constitution and management: WP3
- Resource management services: WP2
  - May be hidden by WP3.

Controller component generates actions recorded and replicated on logs. It requires:

- Log constitution and management: WP3
- Replication and communication mechanism: WP1, WP3
- Resource management services: WP2
  - May be hidden by WP3.

Some **Meeting Management** functionalities require more services:

- Create an event:

  - Create a Telex document: WP3
- Publish an event:
  - Access control services: WP3 (an event is a special file)-WP2
- Import a user document:
  - Get document information (Yellow pages): WP2
  - Import the document: WP3
- Accept /decline mechanism:
  - To vote on someone actions: WP3
  - Decision notification:
    * Communications: WP3, WP1

Similarly, **Collaborator Management** needs:

- Get group list and users' profiles:
  - Yellow pages and group awareness: WP2
- Invite a user to collaborate
  - Access control on Telex Document: WP2, WP3
  - Decision notification:
    * Communications: WP3, WP1
- Stop collaboration with a user:
  - Access control on Telex Document: WP2, WP3
  - Decision notification:
    * Communications: WP3, WP1
- Accept/decline an invitation for collaboration:
  - Access control on Telex Document: WP2, WP3
  - Decision notification:
    * Communications: WP3, WP1

Finally, **Reconciler** component requires:

- Reconciliation and scheduling services: WP3, Semantic Store
- Collaborators' status per Telex document: WP3.

2.3 Use case

Deploying this application using the semantic store layer means supporting optimistic replication. Each user of this application can execute operations locally. However the execution remains tentative until an agreement phase across the participants occurred to agree whether the operations are committed or aborted. If necessary, the agreement phase decides of the order between some concurrent operations.

A simple diagram in figure 2 shows what a concurrent execution with the calendar application might be. Consider users Jean-Michel, Lamia and Marc, using a Grid4All calendar application to plan some meetings between colleagues. Jean-Michel, Lamia and Marc are working separately and communicate only via the application.

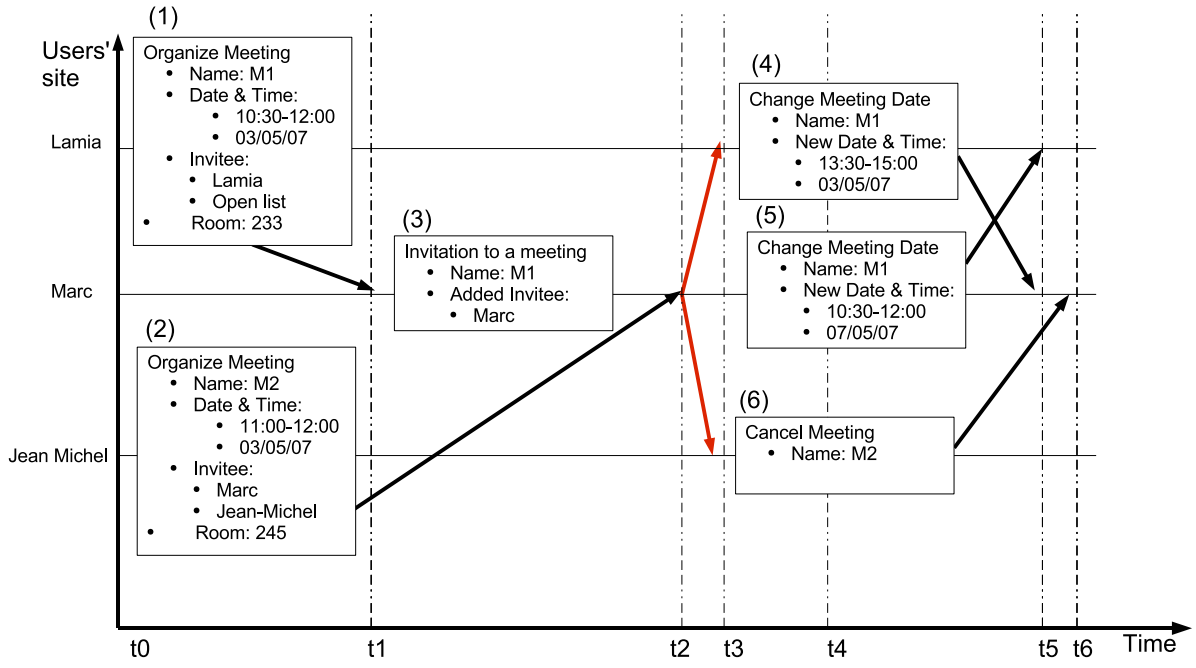


Figure 2: Execution scenario the Shared Calendar application.

Jean-Michel organizes meeting *M2* on *03 of May* between *11:00 and 12:00*. He allocates *Room 245* for that purpose. He requires the presence of *Marc* and himself. Actions (2) in figure 2.

Lamia organizes meeting *M1* on the same day between *10:30 and 12:00*. She allocates *Room 233*. She will attend the meeting, and allows other people to invited themselves. Actions (1) in figure 2.

Suppose that, at some point in time $t1$, Marc has received Lamia's actions, but not yet Jean-Michel's. This may happen, for instance, if Jean-Michel is working offline. Marc is interested in *M1* and invites himself to that meeting. Action (3) in figure 3. Later, at $t2$ Marc knows Jean-Michel's actions.

As *M1* time overlaps with *M2* a conflict is detected and propagated to the concerned users $t3$.

In order to resolve this conflict, Lamia might shift the start time of *M1* to *13:30* (action (4)) Concurrently ($t4$), Marc has set the date of *M1* to the *07th of May* (action (5)), and Jean-Michel has cancelled *M2* (action (6)).

At $t5$ Lamia and Marc have received their concurrent modifications of meeting *M1* . Obviously *M1* is scheduled at different times on Lamia's and Marc's calendar. Shared Calendar application requires from the SS layer to reorder the actions similarly in both sites after the agreement phase.

2.4 Interfacing Shared Calendar to semantic store

This section gives details of the Shared Calendar application based on Telex as Semantic Store. We start describing the semantics of the application using the action-constrain framework ACF [29] in 2.4.1. Then we describe the application data in section 2.4.2. Finally in 2.4.3 present the application architecture and list the pseudo-code using the Semantic Store API.

2.4.1 Shared Calendar semantics

One way to develop this collaborative application is to use the semantic store functionalities.

We consider a shared calendar that supports the following operations:

- *createEvent(meetingId)*: Create some event, for instance a meeting.
- *setInfo (description, time, meetingId)*: Modify the schedule of an event.
- *invite/addUser (userId, meetingId)*: Invite a person to an existing event.
- *allocate (roomId, meetingId)*: Allocate a room for an event
- *cancelInvitation/cancelUser (userId, meetingId)*: Cancel an invitation.
- *cancelAllocation (roomId, meetingId)*: cancel a room allocation for a meeting.
- *Cancel (meetingId)*: Cancel a meeting.

Recall that this application supports optimistic replication. Each user of this application can generate one of the previous actions and execute it locally. However the execution remains tentative until an agreement phase across the participants occurred to agree whether actions are committed, or aborted, or reordered. This commitment process is closely linked to the semantics linking the actions tentatively executed. In the following we call reconciliation protocol, some algorithm ensuring the commitment process.

A simple scenario will give a feel of which semantics can be expressed in the ACF.

A simple scenario Back to the use case described in the section 2.3. Users Jean-Michel, Lamia and Marc plan some meetings using the Shared Calendar application. As said before, Jean-Michel, Lamia and Marc are working separately and communicate only via the SC. Figure 3 shows what their multilog might look like. Recall that a multilog is a data structure recording a set of actions with the semantics constraints between actions.

Jean-Michel organizes *M2* on *03 of May* between *11:00 and 12:00*. He allocates *Room 245* for that purpose. He requires the presence of *Marc*. Thus, he generates the set of actions (a) in figure 3.

Lamia generates a CreateEvent action for *M1* on the same day between *10:30 and 12:00*. She allocates *Room 233*. She generates the set of actions (1) in figure 3.

Jean-Michel wants his actions to act as a parcel, i.e., either all his actions occur jointly, or none occurs. This *atomic* grouping of actions is expressed in the ACF with an *Enables* cycle. Similarly for Lamia's actions.

Suppose that, Marc has received Lamia's actions, but not yet Jean-Michel's. Marc invites himself to *M1*: action (i) in figure 3. Later, Marc knows Jean-Michel's actions.

Setting *M1* time to *10h:30-12:00* and inviting Marc to it is now in conflict with inviting him to *M2* and scheduling *M2* at the same time.

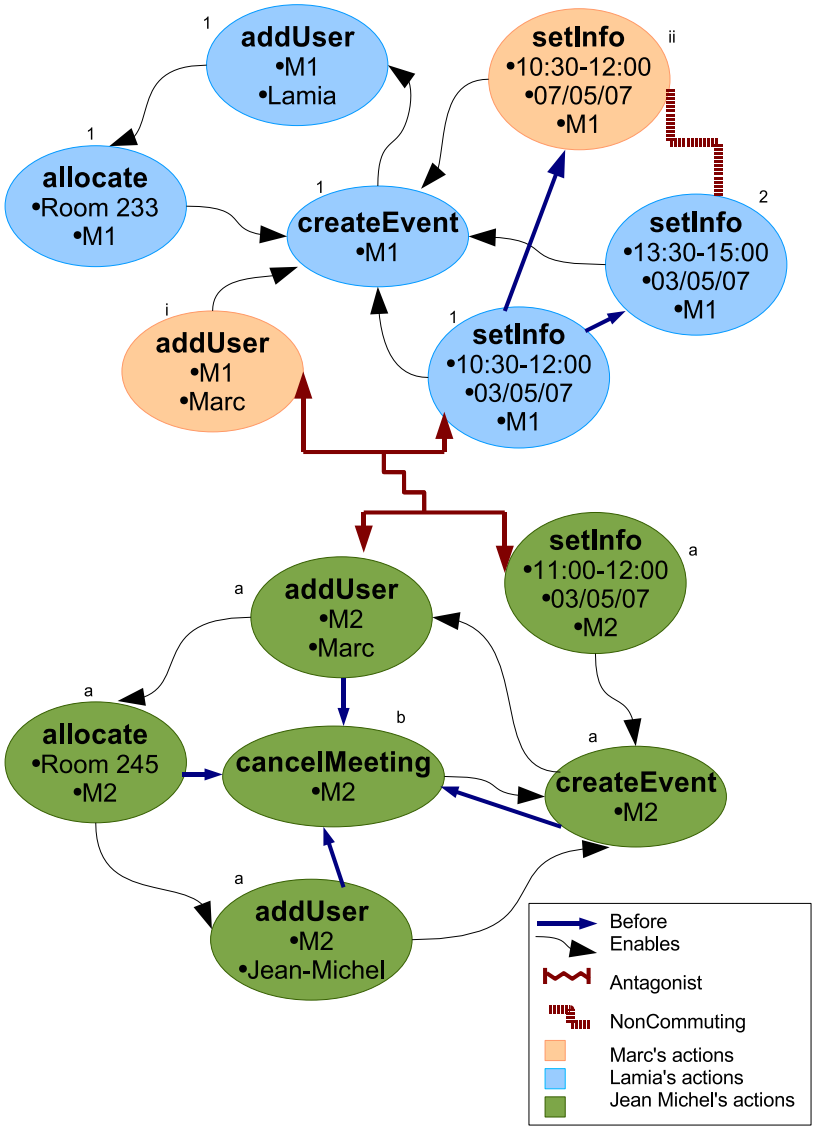


Figure 3: Multilog scenario of a shared calendar application.

In order to resolve this conflict, Lamia might shift the start time of *M1* to *13:30*. Therefor she generates a *setInfo* represented by action (2) in figure 3. In the meantime, Marc has set the date of the *M1* to the *07th of May* and generates action (ii). Jean-Michel has cancelled *M2* by generating action (b).

Marc's and Lamia's modifications (action (2), (ii)) can only be achieved only if the CreateEvent of *M1* succeed. To ensure this condition, there is an *Enables* constraint between the creation and the *setInfo*. Beside, Marc's and Lamia's modifications (action (2), (ii)) have to proceed only after the setInfo (1). To ensure correct ordering, there is a *Before* constraint between these actions.

Similarly, Jean-Michel's cancellation may execute only after his actions organizing *M1* . Besides there is a *dependence* constraint between the cancellation and the creation of the meeting (actions (a), (b)).[4]

This guarantees two things: (i) the actions execute in the correct order at every site, and (ii) if the first action does not execute or fails, then the second action does not execute.

Meanwhile, Lamia and Marc are modifying the same data. Therefore their actions are *non-commuting* with the previous ones. The non-commutation constraint causes the reconciliation protocol to serialise them in the same order at every site.

Note that currently there are useful constraints that ACF cannot express. For instance, a semantics such as "Reserve either Room 245 between 10:00 and 12:00 *or* Room 246 between 14:00 and 16:00" is not currently supported in our framework.

However, when a conflict is set, it can't be removed even if some actions change the state of the calendar. For instance, if one is invited to "M1" at "10:00". Then he gets invited to "M2" at the same time. Thus there is a conflict between his invitation to "M1" and to "M2". If later the date of one of the two meeting changes or is cancelled. The conflict remains and one of the two invitations have to be aborted.

2.4.2 Data type

Documents A user calendar is a telex document. This calendar is shared and can be modified by other users. Thus, this document is a multilog, and is replicated in foreign user's sites.

A created meeting is shared by a varying number of users: the one invited, and can be modified by them in a collaborative way. Thus we made each meeting a Telex document so we can make use of the semantic store functionalities for meeting consistency issues. A meeting document is replicated in all invited user's sites.

As a consequence, a user A possess his own calendar C_A, and may also possess a replica of a user B calendar C_B. Calendar C_A may be replicated to in user B site.

Therefore, the shared calendar application handles multiple separate documents, and multiple replicas of those documents cross multiple sites as figures 4 and 5 show. This figure depict the actions (1) (i) and (a) of the scenario in figure 3 on Marc's and Jean-Michel sites.

Figure 5 shows that Jean-Michel has created *M2* document and has imported a replica of Marc's calendar in order to invite him to that meeting. Beside figure 4 shows that Marc has imported a replica of *M1* to invite him self. He imported a replica of *M2* because of Jean-Michel invitation.

[4] Dependence is the conjunction of *Before* and *Enables*.

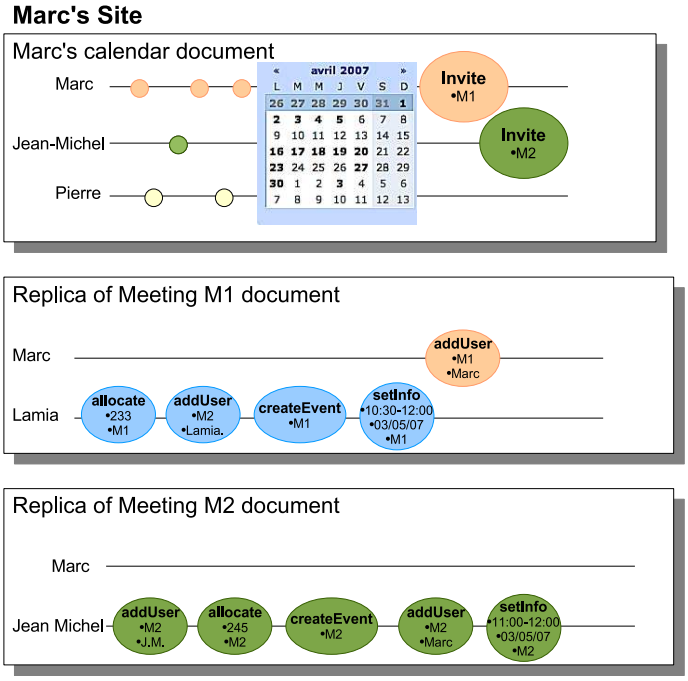


Figure 4: Multiple documents replicated Handled by the Shared Calendar Application on Marc's site.

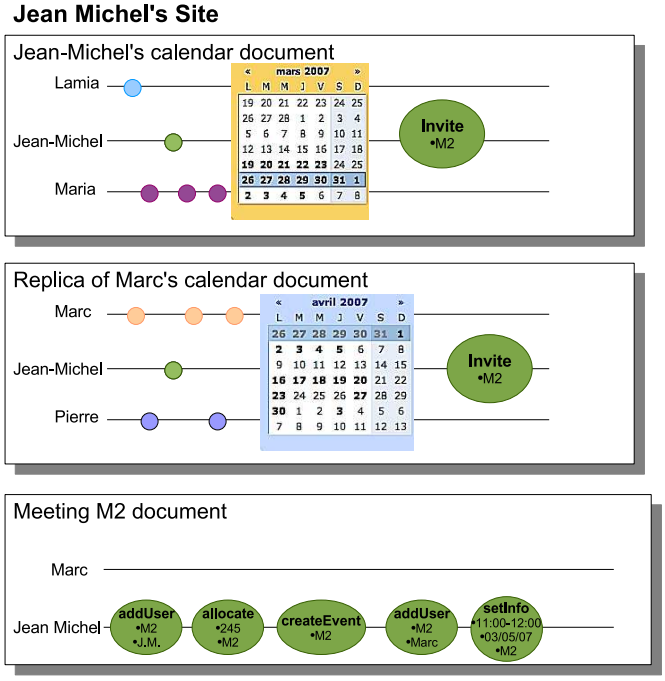


Figure 5: Multiple documents replicated Handled by the Shared Calendar Application on Jean-Michel's site.

Actions In the next sections we use the following notation:

- *userId*: a unique ID of a user. Provide by the VO manager.
- *meetingId*: a unique ID of a meeting event.
- *documentId*: a unique ID of a document.
- *dateId*: time is divided into discrete slots (e.g. 30 min), numbered from an initial date. This sequential number is a unique ID of a precise date.
- *meeting information*: date if not specified elsewhere, resources (room, invited users) and an optional description of a meeting.

From now on, we have seen that the shared calendar application handles multiple documents: calendar documents, and meeting documents. Thus the application supports two classes of actions: actions executed on calendar documents, and actions executed on meeting documents.

When one organizes an event, he creates a meeting document, import the invited calendar document. Notify them of their invitation with an action on their calendar. To set the meeting information, invited users create actions on the meeting document.

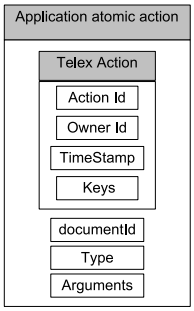


Figure 6: Application atomic action data type.

Figure 6 shows the structure of the application atomic actions as follows:

- Action Id: a unique Id of the action.
- Owner Id: the Id of the user that generates the action.
- TimeStamp: action generation logic time
- Keys: hash of the arguments that Telex use for a quick check whether two actions commute.
- documentId: the identifier of the document on which the action is executed.
- Type: the identity of the action.
- Arguments: the argument needed to execute the action.

In the next paragraphs we list the application atomics actions. we detail their input/output and their structure. Their execution pseudo-code is described in section 2.5.1 and section 2.5.2. We skip the Action Id and timestamp arguments in the description of the action structure.

The calendar application supports the following atomic actions on calendar documents:

***Invite* (userId, meetingId)**

Input:

- userId: the user invited to the meeting *meetingId*;
- meetingId: the concerned meeting.

Output:

- No output.

Structure:

- Owner: get owner Id;
- Keys: check with concurrent CancelInvitation for the same meeting

  ```
  { hash (meetingId +userCalendarId) } ;
  ```
- Document: User calendar document Id;
- Type: “Invite”;
- Arguments:

  ```
  meetingId;
  ```

Description: A user *x* generates *Invite (user1, m)* on *user1*’s calendar document to invite *user1* to the meeting *m*. When *user1* executes this actions, it does the following operations:

1. Import the meeting *m* Telex Document
2. Set the *user1* statuts for the meeting *m* as undefined.
3. Compute meeting *m* data: get and execute a schedule of actions logged on the imported meeting *m* Telex Document.

CancelInvitation **(userId, meetingId)**

Input:

- userId: the user cancelled to the meeting *meetingId*;
- meetingId: the concerned meeting.

Output:

- No output.

Structure:

- Owner: get owner Id;
- Keys: Check with the concurrent invitations for the same meeting.

  ```
  { hash (meetingId +  userCalendarId) } ;
  ```
- Document: User calendar document Id;
- Type: “CancelInvitation”;
- Arguments:

  ```
  meetingId;
  ```

Description: A user *x* generates *CancelInvitation(m, user1)* on *user1*’s calendar document to cancel the invitation of *user1* to the meeting *m*. When *user1* executes this actions, it does the following operations:

1. Ask *user1* if he accepts the cancellation. If *user1* accepts:
2. Get the meeting *m* Telex Document.

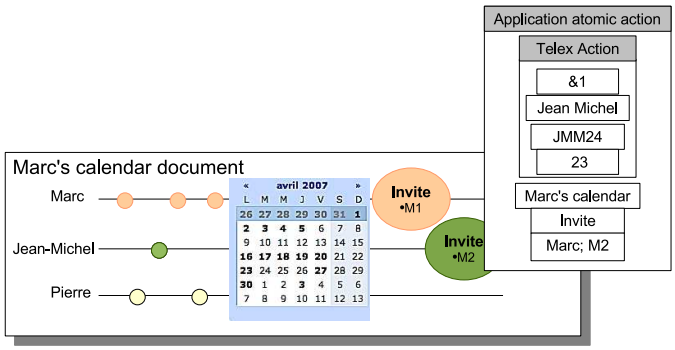


Figure 7: Invite action structure.

3. Set the *user1* statuts for the meeting *m* as cancelled.
4. Close the meeting *m* Telex Document.

Back to the scenario shown in figures 3. Figure 7 shows the structure of the action *invite (Marc, M2)*, generated by Jean Michel and recorded on Marc's calendar.

For the meeting management, the calendar application supports the following atomic actions, they are recorded and executed on meeting documents.

***createEvent* (meetingId)**

Input:

- meetingId: the concerned meeting identifier;
- mTDoc: the corresponding document identifier.

Output:

- meeting object.

Structure:

- Owner: get owner Id;
- Keys: generated only one per meeting. Telex won't have to check, thus the key must be unique.

```
{ hash (meetingId +out of range value) } ;
   // => uniqueness between createEvent of different  meetings.
```

- Document: meetingId Telex document;
- Type: "CreateEvent";
- Arguments:

```
meetingId;
meeting Telex-document Id.
```

Description: A user *x* generates *CreateEvent(m, mTDoc)* on meeting *m*'s Telex-document. When an invited user for meeting *m* executes this actions, it does the following operations:

1. Create a meeting object with a meetingId = *m*, and corresponding Telex document = mTDoc.

***addUser* (userId , meetingId)**

Input:

- meetingId: the concerned meeting identifier;
- userId: the invited user identifier.

Output:

- No output.

Structure:

- Owner: get owner Id;
- Keys: check with concurrent cancelUser/cancelEvent on the same user/meeting.

```
// Check with cancelInvitation actions
Keys = { hash (meetingId + userId) } ;
     // => we will also check with concurrent invitations
     //even if they commute.
```

Check with the cancelEvent

```
Keys += { hash (meetingId + "InviteUser") } ;
```

- Document: meeting document;
- Type: “addUser”;
- Arguments:

```
meetingId;
userId.
```

Description:

A user *x* generates *addUser(m, userId)* on meeting *m*’s Telex-document to invite *user1* to the meeting *m*. When *user1* executes this actions, it does the following operations:

1. Get the meeting object having meetingId = *m*.
2. If *user1* has not accepted the meeting *m*, then:
   (a) Ask him his position.
   (b) Get his vote accept, refuse or wait, as *user1* accepts, refuses or still not decide on meeting *m*.
   (c) Set consequently *user1* status on *m*: “invited” for “accept”, “refuseInvitation” for “refuse”, “undefined” for “wait”.

***cancelUser* (userId , meetingId)**

Input:

- meetingId: the concerned meeting identifier;
- userId: the cancelled user identifier.

Output:

- No output.

Structure:

- Owner: get owner Id;

- Keys: check with concurrent inviteUser on the same user/meeting.

```
Keys = { hash (meetingId + userId) } ;
// => we will also check with concurrent cancellations
 // even if they commute.
```

- Document: meeting document;
- Type: “cancelUser”;
- Arguments:

```
meetingId;
userId.
```

Description: A user *x* generates *cancelUser(m, userId)* on meeting *m*’s Telex-document to cancel the invitation of *user1* to the meeting *m*. When *user1* executes this actions, it does the following operations:

1. Get the meeting object having meetingId = *m*.
2. If *user1* is not already cancelled for meeting *m*, then:
   (a) Ask him his position.
   (b) Get his vote accept or wait, as *user1* accepts or still not decide on meeting *m*.
   (c) Set consequently *user1* status on *m*: “cancelled” for “accept”/ “undefined” for “wait”.

***setInfo* (meetingId, meetingInformation)**

Input:

- meetingId: the concerned meeting identifier;
- meetingInformation: the new meeting information: date, description, . . .

Output:

- modify the meeting object.

Structure:

- Owner: get owner Id;
- Keys:

  Check with concurrent setInfo on the same meeting, check with concurrent cancelEvent on the same meeting.

```
Keys = { hash (meetingId + ‘‘setInfo’’) } ;
```

  Check with any concurrent setInfo for the same user and same set time, on different meeting.

```
 // Set of unique date identifier per slot.
Build dateId setDateId (meeting.date, slot);

For each dateId in setDateId
Keys += { hash (dateId) } ;
```

- Document: meeting document;
- Type: “setInfo”;

– Arguments: clear

```
meetingId;
meetingInformation.
```

Description: A user *x* generates *setInfo (m, mInfo)* on meeting *m*'s Telex-document to set or modify the meeting *m* data. When a meeting *m* invited user *user1* executes this actions, it does the following operations:

1. Get the meeting object having meetingId = *m*.
2. If *user1* his status is undefined for meeting *m*, then notify him with the changes.
3. If he has already accept the invitation, then ask him if he accepts the changes. If he does then:
   (a) Set *user1* free on the previous meeting *m* date.
   (b) Set *user1* busy on the new meeting *m* date.
   (c) Set/modify other informations like meeting description.
   (d) if conflict occurs with other meetings, then notify *user1* and ask him to vote on-conflicting meetings.

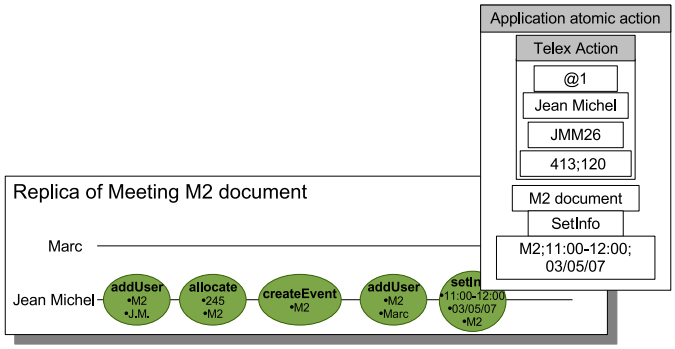


Figure 8: SetInfo action structure.

Back to the scenario shown in figures 3. Figure 8 shows the structure of the action *setInfo (M2, 03/05/07, 11:00-12:00)*, generated by Jean Michel and recorded on M2 document.

***cancelEvent* (meetingId)**

Input:

– meetingId: the concerned meeting identifier.

Output:

– No output.

Structure:

– Owner: get owner Id;

– Keys: check with concurrent inviteUser and setInfo on the same user/meeting.

```
Keys= { hash (meetingId + "InviteUser") } ;
 // => we will also check between all concurrent invitations
  // even if they commute.

 Keys+={ hash (meetingId + "SetInfo") } ;
```

- Document: meeting document;
- Type: “cancelUser”;
- Arguments:

```
meetingId;
userId.
```

Description: A user *x* generates *cancelEvent(m)* on meeting *m*’s Telex-document to cancel it. When a meeting *m* invited user *user1* executes this actions, it does the following operations:

1. Get the meeting object having meetingId = *m*.
2. If *user1* has not already accepts a cancellation for meeting *m*, then:
   (a) Ask him his position on the cancellation.
   (b) Get his vote accept / refuse or wait, as *user1* accepts, refuses or still not decide on canceling meeting *m*.
   (c) Set consequently *user1* status on *m*: “cancelled” for “accept”, “refuseCancellation” for refuse, and “undefined” for “wait”.

***Allocate/cancelAllocation* (roomId, meetingId)** : Allocate/Cancel the allocation of a room for an event. The application has the same behaviour for the room resources and the user resources. A further work will detail these actions. For the next sections we will skip this action.

Constraints As the application manages multiple documents, the constraints can be inside a document or between documents.

In the scenario depicted in figure 3. Lamia generates the following atomic actions (actions (1)):

1. createEvent(M1);
2. allocate (233,M1);
3. setInfo (M1; 03/05/07;11:00,15:00);
4. Invite (M1;Lamia);

These actions are logged in meeting M1 document. To express their atomicity, the application generates an *enables* constraint cycle. As all constrains are between two action on meeting M1, they are also logged on M1 document. Thus this set of constraint is an example of constraint inside a document as shown in figure 9.

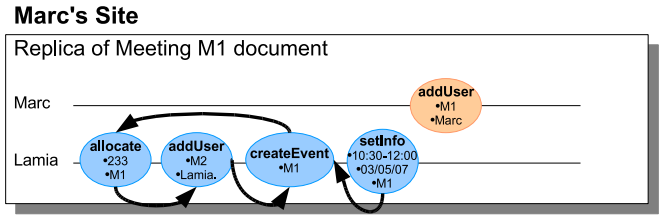


Figure 9: Scenario of constraints inside a document.

Back to the scenario of figure 3. We supposed that, at some point in time, Marc has received Lamia’s actions (action(1)), but not yet Jean-Michel’s. Marc invites himself to *M1*. Action (i) in

figure 3. Later, Marc knows Jean-Michel's actions (action (a)). He thus detect the antagonist constraint between setting *M1* time to *10h:30-12:00* and inviting Marc to it on one side, and inviting him to *M2* and scheduling *M2* at *11h:00-12:00* on the other side.

As this antagonist constraint is between actions on meeting M1 and M2 documents, it is recorded n both documents. Thus this set of constraint is an example of constraint between documents as shown in figure 10.

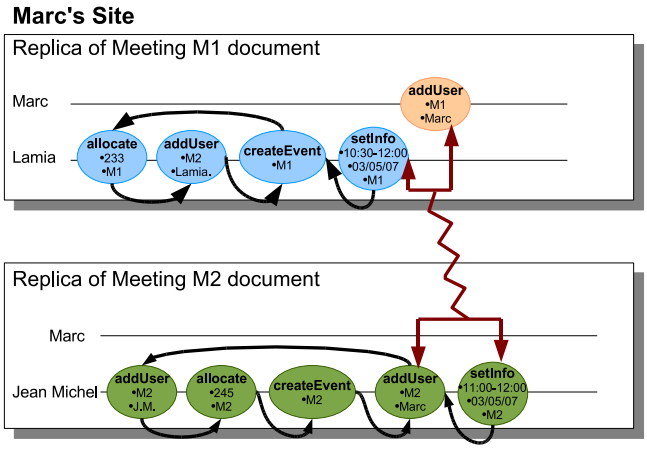


Figure 10: Scenario of constraints between documents.

2.4.3 Application architecture

From now on, we have seen the description of basic actions, and their execution pseudo-code. We have described the two sorts of constraints between those actions. The following section details the architecture of the application and the mechanism to generate, log actions and constraints and execute them.

Figure 11 shows the main component of the application and it's interaction with Telex. A controller gets user operations through the GUI, transform them into a set of basic actions and constraints. Log them in the corresponding document via *addFragment*.

When new action arrives from remote Telex instances, Telex search for possible constraints with the logged actions. A quick check on the constraints filters the commuting actions (keys don't match). Otherwise, Telex asks the application for the corresponding constraints.

The compare component check the arguments of the actions and answers Telex via *getConstraints*.

Finally Telex compute possible schedules and send the to the application via *Execute*. The model component executes the schedule, and the view component shows them to the user via the GUI.

Controller description The controller thread captures the user operation:

- Create meeting;
- Cancel a meeting;
- Invite users;

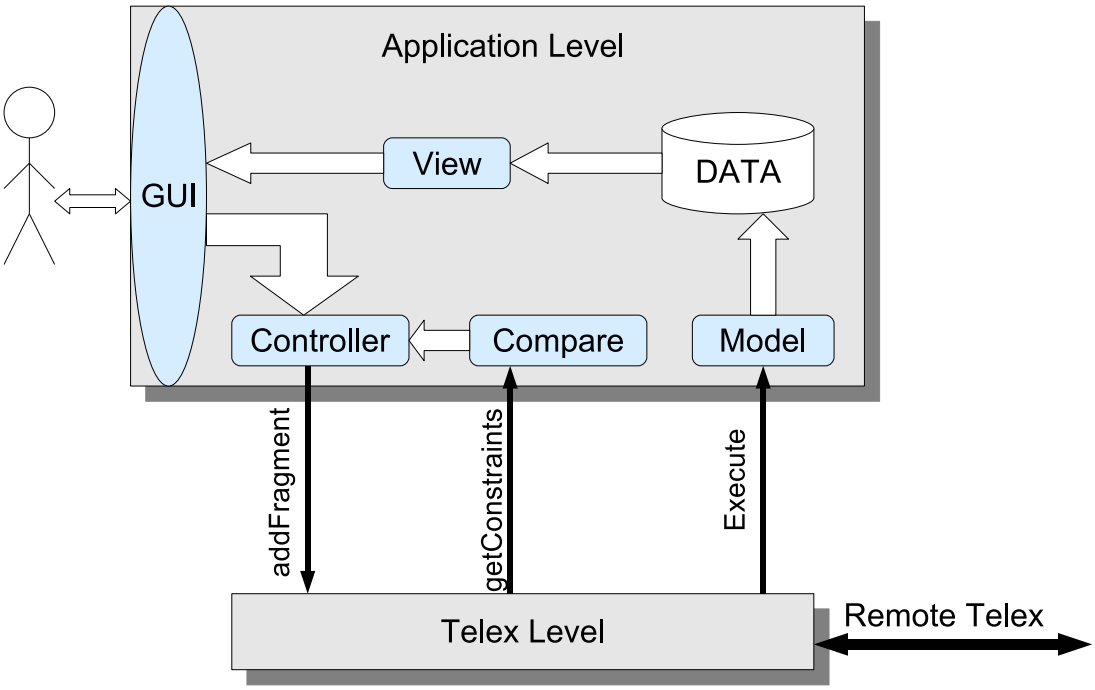


Figure 11: Shared Calendar architecture.

- Cancel invitations;
- Set/modify meeting information: date, description ···

For each event the controller creates consequently actions and constraints. It logs the created actions/constraints in the corresponding Telex document: meeting Telex document or a user calendar Telex-document. Section 2.5.3 shows the executed operations for each event.

Model description Telex computes sound schedules based on the logged actions and constraints. It sends the to the application level via the *Execute* API. The model thread captures telex *Execute* events. Gets the schedules and executes them on a user's calendar and meeting documents.

Section 2.5.4 describes the corresponding pseudo-code.

Compare description The Compare thread handels *getConstraints* events. When new action is logged, Telex search for possible constraints with the previously logged actions. A quick check on the constraints filters the commuting actions (keys don't match). Otherwise, Telex asks the application for the corresponding constraints by rising a *getConstraints* event.

As a key is just a hash of action's arguments, the compare thread has to check real arguments to get the corresponding constraints. It use the correspondence tables 1 2 3 4 5 to get the write constraints . Section 2.5.5 shows the pseudo-code of the compare thread.

In this section, we call *thisUser*, the user on witch site the thread compare is lunched.

We assume that there are no constraints gotten by the *getConstraint* between actions on one calendar document. This is achieved by construction, as the *Invite* and *CancelInvitation* only open the meeting document, and close it.

Table 1: Constraints matching table: createEvent

createEvent	(M_1)	M_1+ *out of range value*		
Action	Arguments	Keys	Real matching condition	Constraints
createEvent	(M_2)	M_2+ *out of range value*	$M_1 = M_2$	$\emptyset$ Same action
X	X	X	$\emptyset$	$\emptyset$

Table 2: Constraints matching table: addUser

addUser	(M_1, U_1)	$M_1 + U_1$; $M_1 +$ $''InviteUser''$		
Action	Arguments	Keys	Real matching condition	Constraints
addUser	(M_2, U_2)	$M_2 + U_2$; $M_2 +$ $''InviteUser''$	$M_1 = M_2$; $(M_1, U_1) = (M_2, U_2)$	$\emptyset$
cancelUser	(M_2, U_2)	$M_2 + U_2$	(M_1, U_1) = (M_2, U_2)	Non-Commuting
setInfo	$(M_2, Date_2)$	$Date_{2Id}$ per slot; $M_2 +$ $''SetInfo''$	$\emptyset$	$\emptyset$
cancelEvent	(M_2)	$M_2 +$ $''InviteUser''$; $M_2 +$ $''SetInfo''$	M_1 = M_2	Non-Commuting

Table 3: Constraints matching table: cancelUser

cancelUser	(M_1, U_1)	$M_1 + U_1$		
Action	Arguments	Keys	Real matching condition	Constraints
cancelUser	(M_2, U_2)	$M_2 + U_2$	(M_1, U_1) = (M_2, U_2)	$\emptyset$
setInfo	$(M_2, Date_2)$	$Date_{2Id}$ per slot; $M_2 +$ $''SetInfo''$	$\emptyset$	$\emptyset$
cancelEvent	(M_2)	$M_2 +$ $''InviteUser''$; $M_2 +$ $''SetInfo''$	$\emptyset$	$\emptyset$

Table 4: Constraints matching table: setInfo

setInfo	$(M_1, Date_1)$	$Date_{1Id}$ per slot; $M_1 +$ $''SetInfo''$		
Action	Arguments	Keys	Real matching condition	Constraints
setInfo	$(M_2, Date_2)$	$Date_{2Id}$ per slot; $M_2 +$ $''SetInfo''$	$M_1 = M_2$	Non-Commuting
			$M_1 \neq M_2$; $\exists slotS$: $Date_{1Id_1} = Date_{2Id_2} = S_{Id}$	Antagonist*
cancelEvent	(M_2)	$M_2 +$ $''InviteUser''$; $M_2 +$ $''SetInfo''$	$M_1 = M_2$	Non-Commuting

*To minimize the number of aborted action:
Get the previous AddUser (M_1, $thisUser$)=A_1
Get the previous AddUser (M_2, $thisUser$)=A_2
Set Constraint (Antagonist, A_1,$setInfo_1$,A_2,$setInfo_2$)
See example page D3.1–Chapter II–33

Table 5: Constraints matching table: cancelEvent

cancelEvent	(M_1)	M_1+ $''InviteUser''$; M_1+ $''SetInfo''$		
Action	Arguments	Keys	Real matching condition	Constraints
cancelEvent	(M_2)	M_2+ $''InviteUser''$; M_2+ $''SetInfo''$	$M_1 = M_2$	$\emptyset$

Back to our example (figure 3), suppose Telex already holds the action *setInfo(M1, 10:30-12:00, 03/05/07)* with keys {hash ("Slot(10:00-11:00, 03/05/07)"), hash ("Slot(11:00-12:00, 03/05/07)"), hash ("M1setInfo"}[5]

When it receives action *addUser(Jean-Michel, M2)* with keys {hash ("M2Jean-Michel"), hash ("M2InvitedUser"} , the latter does not (with high probability) have any common keys with the former. Therefore Telex does not call *getConstraint* with these two actions.

In contrast, when Marc's site receives *SetInfo(M2,11:00-12:00,03/05/07)*, this action definitely has a key in common with the first one (the hash of their common slot (11:00-12:00, 03/05/07)).

Telex calls the *getConstraints* method of the calendar application. The compare thread check the table 4 and:

- gets the previous addUser (M_1, $Marc$) = $addUser^{(i)}$, regarding to *setInfo(M1, 10:30-12:00, 03/05/07)*= $setInfo^{(1)}$.
- gets the previous addUser (M_2, $Marc$) = $addUser^{(a)}$, regarding to *setInfo(M2, 11:00-12:00, 03/05/07)*= $setInfo^{(a)}$.
- Returns conflict between $(addUser^{(i)}, setInfo^{(1)})$ and $(addUser^{(a)}, setInfo^{(a)})$
- Finally record the conflict in M1 multilog and M2 multilog.

[5] Slot = 60mn

2.5 Shared Calendar Pseudo-code

2.5.1 Actions on Calendar Telex-Documents

Invite(m, user1)

```
Invite(m, user1){
Telex Document mTDoc = import (m.Tdoc);
 // set the user status for the meeting m as undefined
userStatus[m,user1]=undefined;
 // get and execute a schedule of the meeting document actions
mTDoc.executeNow();
}
```

CancelInvitation(m, user)

```
CancelInvitation(m, user){
Telex Document mTDoc = getDocument (m);
 // set the user status for the meeting m as cancelled
userStatus[m,user]=cancelled;
mTDoc.close();  // close the meeting m document
}
```

2.5.2 Actions on Meeting Telex-Documents

CreateEvent(m, mTDoc)

```
CreateEvent(m, mTDoc){
 // create the meeting object and associate it with the meetingId m
      //and the corresponding document.
create mObject (m, mTDoc);
}
```

addUser(m, userId)

```
addUser(m, userId){
  // get the meeting object representing the meeting
    //set with meeting information.
MeetingObject mObj= getmObject (m);
if (userStatus[m,userId] == invited) {
vote= accepted;
} else if {
Ask the user ‘‘userId’’ his position on his invitation
 to  the meeting ‘‘mObj’’;
if (vote = accepted) {
userStatus[m,userId]= ‘‘invited’’;
 // set the meeting in the userId agenda
Agenda[userId].addmeeting(m) ;
}
else if (vote = wait)
userStatus[m,userId]= ‘‘undefined’’
else // vote = refuse invitation
```

```
userStatus[m,userId] = ‘‘refuseInvitation’’
}
 // For the reconciliation: tell the collaborators whether
   //  the invitation is accepted, refused or not decided.
actionStatus[ getActionId ] = vote ;
}
```

cancelUser(m, userId)

```
cancelUser(m, userId){
 // get the meeting object representing the meeting
   // set with meeting information.
MeetingObject mObj= getmObject (m);
if (userStatus[m,userId] == cancelled) {
vote= accepted;
} else if {
Ask the user ‘‘userId’’ his position on cancelling his
 invitation  to  the meeting ‘‘mObj’’;
if (vote = accepted) {
userStatus[m,userId]= ‘‘cancelled’’;
 // remove the meeting in the userId agenda
Agenda[userId].removemeeting(m) ;
}
else if (vote = wait)
userStatus[m,userId]= ‘‘undefined’’
else // vote = refuse
                                          userStatus[m,userId] = ‘‘refuseCancellationg’’
}
 // For the reconciliation: tell the collaborators whether the
   // invitation is accepted, refused or not decided.
actionStatus[ getActionId ] = vote ;
}
```

setInfo (m, mInfo)

```
setInfo (m, mInfo){
 // get the userId where the action is executed
getCurrentUser= userId;

if (actionStatus   [getActionId, getUserId] != refused) {
  if (Agenda[userId].get(m.date) == free){
      Agenda[userId].set(m.date, busy, m);
      // if the user didn’t decided on the action then
     // notify him of the change
     if (actionStatus [getActionId, getUserId] == undefined) {
  Ask the user he accepts  the change
  Get vote;
  actionStatus  [getActionId, getUserId] = vote;

  } else {
            // add m to  the meeting list scheduled on m.date
```

```
          Agenda[userId].set(m.date, busy, +m);
  notify the user of  the conflict ;
  get vote;
  actionStatus  [getActionId, getUserId] = vote;
   // =>  conflicting action's votes can be both accepted if
    //  the user resolves the conflict with other setInfo
   // or cancelInvitation
  }
}
```

cancelEvent(m)

```
cancelEvent(m){
 // get the userId where the action is executed
getCurrentUser= userId;

 // get the meeting object representing the meeting
   // set with meeting information.
MeetingObject mObj= getmObject (m);
//Cancelled users automatically accept the cancelEvent
if (userStatus[m,userId] == cancelled) {
vote= accepted;
} else if {
Ask the user ''userId'' his position on cancelling the meeting
 to  the meeting ''mObj'';
if (vote = accepted)
userStatus[m,userId]= ''cancelled'';
 // remove the meeting in the userId agenda
Agenda[userId].removemeeting(m) ;

else if (vote = wait)
userStatus[m,userId]= ''undefined''
else // vote = refuse
userStatus[m,userId] = ''refuseCancellationg''
}
 // For the reconciliation: tell the collaborators whether
   // the invitation is accepted, refused or not decided.
actionStatus[ getActionId ] = vote ;
}
```

2.5.3 Controller thread

On a create meeting event

```
  get owner;
 //all needed users but the owner
get neededInvitedUserList;
 //other invited users
get optionalInvitedUserList;
//Precondition : neededInvitedUserList INTER optionalInvitedUserList
```

```
      // INTER {owner} = NULL

invitedUserList =  neededInvitedUserList +
                           +optionalInvitedUserList
                                        + owner;
get description;
get date;

 // Create the meeting document m

CreateDocument mTDoc;
Meeting m= new Meeting ( owner, InvitedUserList, description,
                                                    dateIdSet...);

Fragment meetingFragment, calendarFragement;
  // Generate the createEvent action
  Action create =  new createEvent (m);
  meetingFragment.add( create);

  Action addUserAction, invite;
  Document di;

  //  For each invited user
  for (user in  invitedUserList){

        // Import user calendar
        di=getdocument (user, calendar);

        // Generate action Invite (m, user) on that calendar
        invite=meeting.createInviteAction(m, user);
        calendarFragment.add (invite);
        di.addFragment (calendarFragment);

        //Generate action addUser(user) to  mTDoc
        addUserAction = new action(m,addUser, user);

        meetingFragment.add (addUserAction);

  }

   // if the meeting can't take place if some needed users can't come
   // => Generate the parcel between:
        // createEvent and addUser (neededUser) set;
meetingFragment.addConsraint (parcel, create,
                              set of addUser (neededInvitedUserList) );

 // In all case the addUser(optionalInvitedUser) need
        // the createEvent action
for (user in optionalInvitedUserList){
```

```
meetingFragment.addConstraint (Enables, addUser(user), create);
}

 //Set the meeting information
     // Generate the setInfo action
     // this action depends on the createEvent

Action setInfo = new action (m,description, date...);
meetingFragment.add(setInfo);
meetingFragment.addConstraint(Enables, setInfo, create);

//Order the invitation with the concurrent invitations
meetingFragment.addConstraint(Before, setInfo, F.getActions(AddUser));

 // Add the fragment to  the meeting document
mTDoc.addFragment (meetingFragment);

 //Ask for a schedule
executeNow();
}
```

On an cancel meeting event

```
get Meeting m;
get owner; // the owner of the action

 // Create cancelEvent action on the meeting document
 Fragment meetingF;
Action cancelEvent= new action(CancelEvent, m);
meetingFragment.add (cancelEvent);
 // If committed it's execution generates the cancelEvent
//in user's Calendar

//get the createEvent action and add the enables constraint
Schedule sc = m.getCurrentSchedule();
Action create = sc.getAction (createEvent);
meetingFragment.addConstraint (Enables, CancelEvent, create);

 //  add ordering constraint (Before) with the previous addUsers
  // and setInfo of mTdoc
meetingFragment.addConstraint (Before,sc.getActions(addUser) ,
                                      cancelMeeting );
meetingFragment.addConstraint (Before, sc.getActions (setInfo));

 // add the fragment on the meeting document
m.getDocument().addFragmet (meetingFragment);
```

```
 //Ask for a schedule
executeNow();
```

On an invite users event

```
get Meeting m;
get owner; // the owner of the action

get neededInvitedUserList; //all needed users
get optionalInvitedUserList; //other invited users
//Precondition : neededInvitedUserList INTER optionalInvitedUserList = NULL
//Precondition : Invited UserList are note already collaboration
 // on the meeting document

//NOTE: owner may be in neededInvitedUserList

invitedUserList =  neededInvitedUserList + optionalInvitedUserList;


// get the createEvent action for the Enables constraints
// the setInfo actions, cancelUsers actions for the ordering constraints

Schedule sc= m.getCurrentSchedule();
Action create =sc.getAction (createEvent);

Fragment meetingFragment, cancelInvitationFragment, setInfoFragment;
//the last concurrent meeting modification
setInfoFragment= sc.getActions (setInfoAction);


Action addUserAction, invite;
Document di;
Document mTDoc = m.getDocument();

 //  For each invited user
  for (user in  invitedUserList){

Fragment calendarFragment = new (Fragment);
        // Import user calendar
        di=getdocument (user, calendar);

        // Generate action Invite (m, user) on that calendar
        invite=meeting.createAction(Invite, m,user);
        calendarFragment.add (invite);

 // order it with the previous cancellations on the calendar document
```

```
cancelInvitationFragment = di.getSchedule().getActions(CancelInvitation);
calendarFragment.addConstraint (Before, cancelInvitationFragment, invite);

//record the calendarFragment in user calendar Document;
di.addFragment (calendarFragment);

        //Generate action addUser(user) to  mTDoc
        addUserAction = new action(m,addUser, user);
        meetingFragment.add (addUserAction);

// Generate Constraint Enables with the createEvent action.
meetingFragment.addConstrainte (Enables, addUserAction, create);

//order the generated AddUser with previous cancelUsers and setInfo
   // get the scheduled  cancelUser , may be multiple cancelling
cancelInvitationFragment=  sc.getActions(cancelUser(user));
meetingFragment.addConstraint (before, cancelInvitationFragment, addUserAction);

  // order with concurrent  setInfo actions, may be multiple.
   meetingFragment.addConstraint (Before, setInfoFragment,
     addUserAction);

  }


// If the owner presence depend on the presence of some neededUser
// and if the owner invitation status is not decided
  //  => then add the enables constraint between last
  // recorded addUser (owner) action
  // and the last addUser(neededUser)
  // Note that both can either be already recorded in the meeting document
  // or generated by the previous loop

if (userStatus[m,owner] =undefined){
for (usr in neededInvitedUserList){
meetingFragment.addConstrainte (Enables,
 getLastAction(addUserAction(owner)),
  getLastAction(addUserAction(usr))
          );
   }
}

// Add those information in the meeting document
mTDoc.addFragmet (F);
  // Ask for a schedule
executeNow();

}
```

On an cancel user's invitation event

```
get Meeting m;
get owner; // the owner of the action

get removedUserList; //all cancelled users
//Precondition : removed users are already collaborating
 /on the meeting document
//NOTE: owner may be in removedUserList


// get the createEvent action for the Enables constraints
// the setInfo actions, cancelUsers actions for the ordering constraints

Schedule sc= m.getCurrentSchedule();
Action create =sc.getAction (createEvent);

Fragment meetingFragment, invitationFragment;
Action addUserAction, invite;
Document di;
Document mTDoc = m.getDocument();

 //  For each cancelled user
  for (user in  removedUserList){

Fragment calendarFragment = new (Fragment);
        // Import user calendar
        di=getDocument (user, calendar);

        // Generate action cancelInvitation (m, user) on that calendar
        cancelInvitation=meeting.createAction(CancelInvitation, m,user);
        calendarFragment.add (cancelInvitation);

 // order it with the previous invitations on the calendar document
invitationFragment = di.getSchedule().getActions(Invite);
calendarFragment.addConstraint (Before, invitationFragment,
                           cancelInvitation);

//record the calendarFragment in user calendar Document;
di.addFragment (calendarFragment);

        //Generate action cancelUser(user) to  mTDoc
        cancelUserAction = new action(m,cancelUser, user);
        meetingFragment.add (cancelUserAction);

// Generate Constraint Enables with the createEvent action.
meetingFragment.addConstrainte (Enables, cancelUserAction, create);

 //order the generated cancelUser with previous addUsers and setInfo
```

```
   // get the scheduled  cancelUserInvitation , may be multiple cancelling
invitationFragment=  sc.getActions(addUser(user));
meetingFragment.addConstraint (before, cancelInvitationFragment,
    //cancelUserAction);

  }


// Add those information in the meeting document
mTDoc.addFragmet (F);
  // Ask for a schedule
executeNow();

}
```

On a modify meeting information event

```
get Meeting m;
get owner; // the owner of the action

get date;
get description;

get neededUserList; // needed users to  execute the modifications
//Precondition: must already collaborate on the meeting

// get the createEvent action for the Enables constraints
// the setInfo actions for the ordering constraints

Schedule sc= m.getCurrentSchedule();
Action create =sc.getAction (createEvent);

Fragment meetingFragment,  setInfoFragment;

Document mTDoc = m.getDocument();

// Generate the setInfo action and record it in the meeting document
Action setInfoAction = createAction (setInfo, m, date, description);
meetingFragment.add (setInfoAction);

// Generate Constraint Enables with the createEvent action.
meetingFragment.addConstrainte (Enables, setInfoAction, create);

//  get the last concurrent meeting modification
setInfoFragment= sc.getActions (setInfoAction);
// order with the previous setInfo actions
meetingFragment.addConstraint (Before, setInfoFragment, setInfoAction);
```

```
 // If the owner wants to  apply the modification only if some neededUsers
   // accepts to  attend the meeting.
 //  For each needed user
  for (user in  neededUserList){

         // get the Last scheduled  addUser
          // may be multiple addUsers, choose the one after the last cancelling
Fragment invitationFragment=  sc.getLastActions(addUser(user));

// set the Enables constraint
meetingFragment.addConstraint (Enables, setInfoAction, invitationFragment);
  }

// If the owner presence depends on this modification
  //  If he wants to  notify the others ,
// and if the owner invitation status is not decided
  //  => then add the enables constraint between last recorded
   // addUser (owner) action and the setInfo

if (userStatus[m,owner] =undefined){
 // get the Last scheduled  addUser (owner)
 Fragment invitationFragment=  sc.getLastActions(addUser(owner));

meetingFragment.addConstrainte (Enables, invitationFragment, setInfoAction );
}

 // Add those information in the meeting document
mTDoc.addFragmet (F);
  // Ask for a schedule
executeNow();

}
```

2.5.4 Model thread

On a Execute event

```
get schedules sc;
// the schedules may be on multiple documents: meetings and user calendar
for (each document d) {
get the initialState;
Restore the initialState;
Schedules subSc = sc.getsubschedule (d );

for (each Action act in subSc) {
// execute the action as described
  }
```

```
execute (act);
}
}
```

2.5.5 Compare thread

On a getConstraints event

```
 //get action to  compare
get action1, action2;

// get the corresponding constraint from the constraint tables.
Fragment constraintFragment = correspondanceTable[action1, action2];

//record the constraints in the corresponding action documents
if (getDocument(action1) != getDocument(action2)) {
getDocument(action2).addFragment( constraintFragment );
}
getDocument(action1).addFragment( constraintFragment );
```

3 Collaborative Editors

3.1 State of the art

A wiki is a collaborative editor on the web: several users can edit common data from distributed sites on the Internet. This section briefly describes the state of the art of collaborative editors and wikis.

3.1.1 Collaborative Editors

We focus on collaborative editors based on optimistic replication (which is the focus in Grid4All). Optimistic data replication allows users, even disconnected, to access data quickly. and support simultaneous editions. The system is then in charge of ensuring the convergence of the updated data towards the best possible state.

The collaborative editors that can support the management of optimistic replication are IceCube, Bayou and WOOT. These approaches ensure the consistency of the data in a state that takes into account the modifications of the users. However, P2P networks are composed of a very volatile population. Each node can enter or leave constantly, and arbitrarily quickly. In this context, the existing solutions of the collaborative editors cannot be adapted.

IceCube is a general-purpose reconciliation system that exploits the application semantics to resolve conflicting updates [12, 27, 28]. In IceCube, update operations are called actions and they are stored in logs. IceCube captures the application semantics by means of constraints between actions, and treats reconciliation as an optimization problem where the goal is to find the largest set of actions that do not violate the stated constraints.

Bayou is a mobile database system that lets a user replicate a database on a mobile computer, modify it while disconnected, and synchronize with any other replica of the database that the user happens to find [34, 22]. In Bayou, each operation has attached a dependency check and a merge procedure. The dependency check is run to verify if the operation conflicts with others whereas the merge procedure is executed to repair the replica state in case of conflict. In Bayou, a single primary site decides which operations should be committed or aborted and notifies other sites about the sequence in which operations must be executed. Anyway, Bayou remains different from single-master systems as it allows any site to submit operations and propagate them, letting users to quickly see the operations effects. In single-master systems, only the master can submit updates.

OT (Operational Transformation) was developed for collaborative editors [9, 32, 30, 31, 35]. OT assumes that a user applies commands immediately at the local site, and then propagates these commands to other sites. As a result, all sites perform the same set of operations but possibly in different orders. The goal of OT is to preserve the intention of operations and assure replica convergence. This is achieved by defining for every pair of concurrent operations a rewriting rule. In [20] it is proved the correctness of OT for a shared spreadsheet. Molli and all [16] extend the OT approach to support a replicated file system. Ferrié and all [10] deal with undo operations

in the context of OT by providing a general undo algorithm based on the definition of a generic undo-fitted transformation.

WOOT [17] proposed by ECOO team of the LORIA, can also be used in a decentralized way. WOOT is based on the monotonous calculus of the linear extension of the order partial between the elements of a linear structure. To integrate each modification, each site is able to make independently this calculus in a polynomial time. WOOT uses neither state vectors nor sites having particular tasks or constraints.

Examples of collaborative editors are [6]

ACE is a simple text editor with standard features such as copy/paste and load/save.Multiple documents can be edited at the same time. Furthermore, ACE can share documents with other users on different computers, connected by communication networks (LAN, Internet).ACE allows the discovery of users and moreover automatically shares their documents.Users can opt to join any discovered shared document.For all of this, no configuration is necessary since it is a zero-conf networking approach(also known as Bonjour or Rendezvous).

ACE builds upon open technologies such as BEEP (RFC 3080) and zero-conf networking applications that understand the public protocol of ACE. ACE is a free software, running on all of major operating systems (Windows, Mac OS X and Linux).

The heart of the application is a concurrency control algorithm based on Operational Transformation [33], which allows lock-free edition of documents by multiple users. It does not impose editing constraints and solves conflicts automatically. designing and implementing.

Intelligent Collaboration Transparency (ICT) is an application sharing framework for sharing familiar single-user tools (applications) for collaboration purposes without modifying their source code. At the user level, unmodified heterogeneous applications can be shared and interoperated. At the system level, the application sharing middleware is able to understand the behavior of the applications being shared. The main assumption underlying this work is that allowing collaborators to use familiar single-user tools for cooperative work can reduce development, deployment, and learning costs while improving individual and group productivity.

CoWord (Microsoft Windows) is a collaborative software component which converts Microsoft word into a real-time collaborative word processor and allows multiple users to collaboratively edit the same word document at the same time.

DocSynch is a collaborative editing system on top of IRC (Internet Relay Chat). By transforming single-user editors into multi-user editors, it allows to remotely edit text documents together. Implementations are targeted as extensions to many popular text editors and IDEs. A working version is available for jEdit.

[6] http://en.wikipedia.org

CodeWright is rich featured programmer's editor from Borland. Its built-in CodeMeeting plugin allows each file to be edited by one person at time. Others can watch cursor actions. Chat is also provided. CodeWright is considered easy to use by many developers and has powerful extension tools. The program can be configured to work with other integrated development environment (IDE) systems, synchronise with IDEs on the fly.

Writeboard is a free collaborative (but not real time) text editor, which allows creation of an unlimited number of web-based text documents. Each Writeboard has a separate user name and password, and changes can be monitored via an RSS feed. Writeboard is a very simple application but supports Diff (file comparison utility), allowing users to compare changes made to the document.

3.1.2 Cooperative editing system, general issues

A real-time cooperative editing system such as ACE allows multiple users to view and edit the same document at the same time from geographically dispersed sites. The following requirements have been identified for such systems:

- Real-time: The response to local user actions must be quick, ideally as quick as a single user editor, and the latency for reflecting remote user actions is low (determined by external communication latency only).
- Distributed: Cooperating users may reside on different machines connected by communication networks with non-deterministic latency.
- Unconstrained: Multiple users are allowed to concurrently and independently edit any part of the document at any time, in order to facilitate free and natural information flow among multiple users.

A real-time cooperative editing system consists of n instances, each instance run by a different user. All instances are connected by a network. One of the most significant challenges in designing and implementing real-time cooperative editing systems is consistency maintenance among the different document replicas (one for each site). A cooperative editing system is said to be consistent if it always maintains the following properties:

- Convergence: Guarantees when the same set of operations (for example, an operation can be an insert/delete of a character) have been executed at all sites, that all copies (also known as replicas) of the shared document are identical. That is, this property ensures the consistency of the final results at the end of a cooperative editing session.
- Causality preservation: For any pair of operations Oa and Ob, if $Oa \rightarrow Ob$ (that is, Ob is dependent on Oa), then Oa is executed before Ob at all sites. That is, this property ensures the consistency of the execution orders at all sites of dependent operations during a cooperative editing session.
- Intention preservation: For any operation O, the effects of executing at all sites are the same as the intention of O, and the effect of executing does not change the effects of independent operations. That is, this property ensures that executing an operation at remote sites achieves the same effect as executing this operation at the local site at the time of its generation, and the execution effects of independent operations do not interfere with each other.

3.1.3 Wiki

A wiki is a website that allows users to add, remove, and edit content. Thus, using web technology, a wiki provides an easy and cheap way to foster mass collaborative authoring. The first wiki (WikiWikiWeb) was developed by Ward Cunningham in the mid-1990s. Wikis allow for linking among any number of pages over the web. This ease of interaction and operation makes a wiki an effective tool for mass collaborative authoring. Wikipedia, an online encyclopedia, is one of the best known wikis.

A wiki software [7] is a type of collaborative software that runs a wiki system. This typically allows web pages to be created and edited using a common web browser. Examples of wiki systems are:

XWiki (second generation wiki) is a Java wiki engine with a complete wiki feature set (version control, attachments, etc.) and a database engine and programming language which allows database driven applications to be created using the wiki interface.

Clearspace is a commercial J2EE application, made by Jiva Software, which combines wiki, blog, and document management functionality into a complete enterprise collaboration solution. Clearspace uses wiki-style markup or WYSIWYG editing to allow for clean version control and workflow management.

Corendal Wiki is a GPL application for corporate environments, with tight integration with Microsoft Active Directory.

OpenWiki is written in VBScript, uses the ASP protocol, and stores data in XML files or Microsoft SQL Server. It combines useful features of several Wiki engines with windows integrated authentication so users are logged in transparently.

TWiki is a structured wiki, typically used to run a project development space, a document management system, a knowledge base, or any other groupware tool.

3.2 XWiki P2P

3.2.1 Requirements of Collaborative Applications

With the popularity of distributed applications and growing speed of internet, collaboratively creating and managing information has become an essential requirement for the success of virtual organizations. In this section we discuss requirements of collaborative applications.

Requirements

[7] http ://www.wikipedia.org

Distributed collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, peer-to-peer, and mobile computing). As an example of such applications, consider a second generation Wiki that works over a peer-to-peer (P2P) network and supports users on the elaboration and maintenance of shared documents in a collaborative and asynchronous manner. Consider also that each document is an XML file possibly linked to other documents. Therefore, such a Wiki allows collaboratively managing a single document (e.g. a scientific paper shared by a few of authors) as well as composed, integrated documents (e.g. an encyclopedia or a knowledge base concerning the use of an open source operating system). Although the number of users that update in parallel a document d is usually small, the size of the collaborative network that holds d in terms of number of nodes may be large. For instance, the document d could belong to Wikipedia, a free content encyclopedia maintained by more than 75,000 active contributors [8].

Many users frequently need to access and update information even if they are disconnected from the network, e.g. in an aircraft, a train or another environment that does not provide good network connection. This requires that users hold local replicas of shared documents. Thus, a P2P Wiki has need for multi-master replication to assure data availability at anytime. In the multi-master approach, updates made offline or in parallel on different replicas of the same data may cause replica divergence and conflicts, which should be reconciled.

In order to resolve conflicts, the reconciliation solution can take advantage of application semantic illustrated in Figure 12. This example deals with a single document elaborated by three authors. The document is a scientific paper organized as a tree. Each node (element) in the tree structure corresponds to a section of the paper and holds the name of the responsible author. Figure 12 (a) shows the initial structure of the paper whereas Figure 12 (b) shows conflicting updates (in gray) over the initial structure. In Figure 12 (b) Esther tries to move the Background section under Paper thereby changing the Background path from *Paper/Solution/Background* to *Paper/Background* while Manal tries to insert two topics under Background using the path Paper/Solution/Background. If the move operation is accomplished before the insert operations, the Background's path changes so that the insert operations do not find the Background element, and therefore such inserts are lost. We can automatically solve this problem by introducing the following application semantic: update operations precede move operations. In Figure 12 (a) according to this semantic, Topic 1 and Topic 2 are inserted in the path *Paper/Solution/Background*, and then the entire sub tree under Background is moved in such a way that the intents of both users (Esther and Manal) are preserved.

In Figure 12 (a), another conflict takes place if Vidal tries to delete Background while in parallel Manal tries to update the contents associated with Background. In this case, it is impossible to preserve the intents of both users as we previously did, i.e. an operation will be preserved and the other one will be discarded. By taking into account the application semantic, we can preserve the operation that would likely be held by the users; in contrast, if we do not consider the application semantic, either we keep this conflict to be manually solved later or we randomly resolve the conflict. Thus, in order to automatically behave as users would likely do, we introduce the following application semantic: ancestral responsible has higher priority than descendent responsible. For instance, according to this semantic, the deletion of Background would be preserved and its update would be discarded since Vidal, who proposes the deletion, is ancestral responsible wrt.

[8] http ://www.wikipedia.org

Manal (i.e. Vidal is responsible for an element in the tree - the Solution element - that is Background's ancestral). As in the real world, we take advantage of the authors' hierarchy to decide conflicts. Of course, sometimes it is better to preserve the operation submitted by the descendent responsible. To cope with this situation, we improve our application semantic as follows: it is possible to reapply discarded updates if the priority-based resolution is not satisfactory. Such semantic can be easily implemented by allowing users to retrieve the discarded operations and try again to execute some of these operations, if they want.

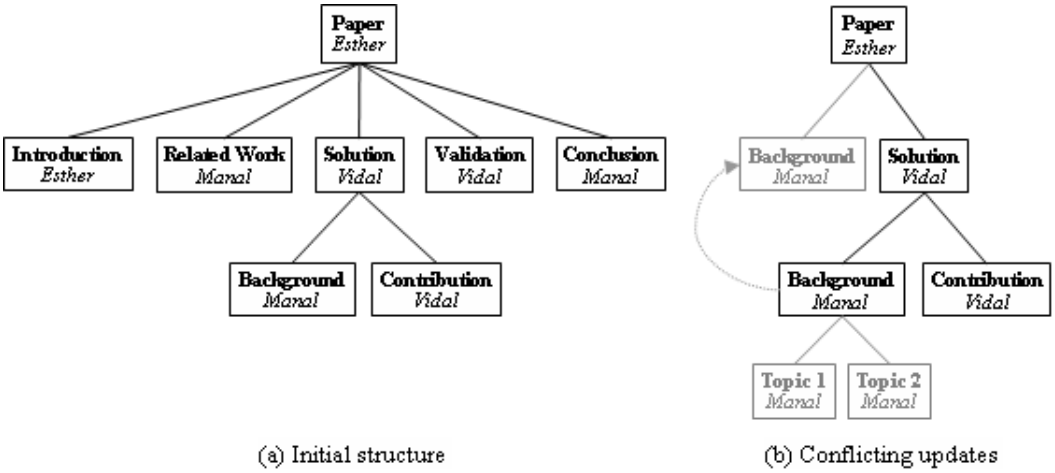


Figure 12: Producing a paper in a collaborative manner

The semantic associated with a P2P collaborative editor can be richer than the simple semantic that we discussed. However, we made the example deliberately simple only to show that, by taking advantage of the application semantic on the reconciliation, we can eliminate spurious update conflicts (e.g. insert and move operations over the same element are not really conflicting operations) and we can resolve the real existing conflicts in an automatic manner as users would likely do.

To manage information, users take advantage of different devices such as notebooks, PDAs and portable phones, which can be supported by networks of variable quality. As a result, it is not acceptable that the replication solution make strong assumptions about the network.

3.2.2 Detailed Examination of XWIKI

We use XWiki as collaborative application to validate Semantic Store reconciliation. In this section, we present the current structure of XWiki application, which is client server. Then we discuss the extensions needed to do a P2P version of XWiki.

XWiki

A wiki is a website that allows users to edit, add, and delete contents. XWiki is second generation of wiki. It is an open source wiki written in java. It provides the editing, attachment, different skins, and full text searching facility to its users. It has also some advanced features; we can export our data from XWiki pages to .pdf pages. It uses hibernate for storing data in relation database and for accessing it. We can use XWiki in different languages. XWiki is a user-friendly

wiki. User can edit or create the page on just single click. For editing, it gives two basic choices general mode and advanced mode. General mode is for basic or normal users and for expert users it gives advanced mode. User can change profile, picture, or password. For editing we can use GUI or direct programming language of XWiki i.e. velocity and groovy.

Any change in document is saved under version control of XWiki feature. We can access the old documents by clicking the history button on the page. An XWiki page can contain attachments, which can later be referenced from within a page. For searching, it provides a full text-searching panel. If a user has administrator rights after searching, he/she can be delete the page and assign specific rights of page to another user.

We can use different kinds of plug-ins with XWiki. One very user-friendly feature of XWiki is Skins. We can use different kinds of skins according to our desire and can make our XWiki page more nice and beautiful.

Current Design

XWiki is based on open source projects; it uses different kinds of open source projects, i.e.

- **Hibernate:** for accessing and working on data;
- **DBCP:** database connection pool for establishing connection with database;
- **JRCS:** used for version of documents; Struts: XWiki uses struts framework for data presentation and adding;
- **OScahe:** used for caching the documents;
- **OSUser:** used for user authentication on system.

Currently XWiki is a client server application as shown in Figure 13. Administrator on Server creates an account for a user and user can access it through username and password using a web browser. Administrator creates a group of users and assign specific rights to that group of users that what kind of work they can do on documents of their XWiki. They either can do editing or deleting or can add new documents and information into their XWiki.

XWiki uses hibernate to access and retrieve data from database. Hibernate is high performance object ,query service and a mapping tool for java environment. Hibernate has three essential components and some optional components. Essential components are session factory, session, transaction. XWiki uses RDBMS; we can use HSQL, MY SQL, or any other relational database.

Scripting

XWiki uses groovy and velocity as scripting languages. Velocity is used as a carrier of data between java layer and page or presentation layer. The programmers of the XWiki core have gathered objects of various types and placed them in the Velocity context. These objects, and their methods and properties, are accessible via template elements called references and effectively form an API for XWiki. Groovy mainly used by administrator or those users who have rights for programming on XWiki. Mainly it is used for integration of plug-ins or components with XWiki. It generates XML. HTML is used at the page layer for presentation of the data. With the use of

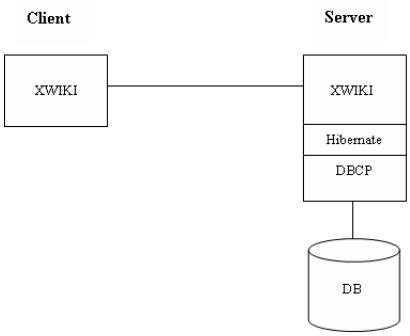


Figure 13: XWiki Client Server

groovy and velocity we can build or write directly in XWiki without compilation. These are easily integrates in J2EE and a programmer can build application fast.

Concept of conflict

In current client server XWiki user can create a document and other users who have rights can easily access the document and can do work on it. From Figure 14 we can see that, if the owner of page edits the page and at same time another user edits the page, owner has priority for editing the page. Owner edit the page and write c ,other user also work on same page and write d, they both save the page at same time. Owner of the document changes will be saved. While in another case if two different users that both are not owners of the page are working together on the same page the last user will have priority, who will update the page in last.

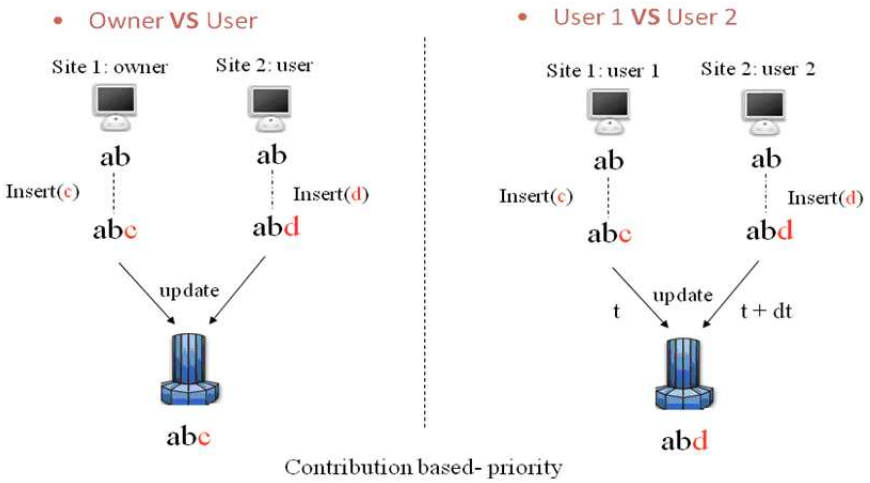


Figure 14: XWiki confliction

Granularity

The current granularity level of update in XWiki is its specific page, where we can edit, or updating information or contents of the page. For XWiki P2P it should be $LINE$ level.

Architecture of XWiki Document

From Figure 15 we can see that XWiki has different spaces (e.g. blog, news, calendar etc) and every space has its own pages. A registered user can access the space and its pages and create, edit the pages in different spaces if user has rights to do edition or creation.

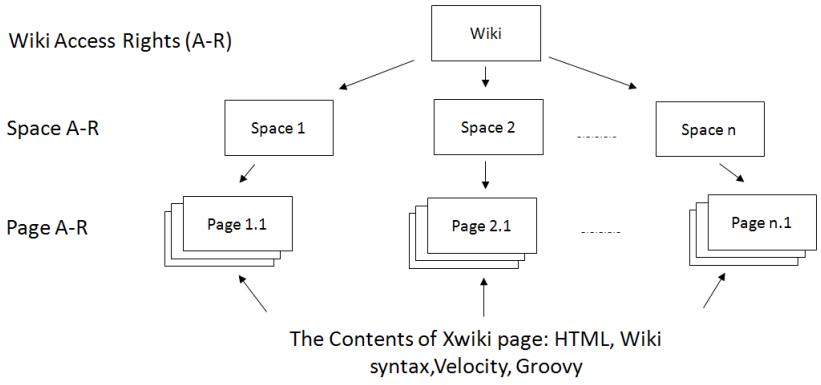


Figure 15: Architecture of XWiki Documents

3.2.3 XWiki P2P

XWiki has client-server architecture. Thus, the centralized server can be a bottleneck and a single point of failure. To improve XWiki data availability, in case of failure or offline mode of server, we propose to make Xwiki peer-to-peer (P2P). The main advantage of making XWiki P2P is availability of its data at any time. With XWiki P2P, a user with the proper access rights is able to access any other user's data and then use the data. Another advantage of P2P is scalability: it is easy to add new peers in the system since administration can be distributed. Finally, XWiki P2P does not require any new infrastructure and can simply use the peers' computing resources, without needing any particular server. To make XWiki P2P, we have the following requirements.

Scalability

Resource management is not an issue in small or medium sized networks. However, when we want to manage different kinds of software's, a large number of users, and disparate users and service availability requirements, scalability is a major concern. Scalability is a property of a system that can handle successfully growing amounts of work and of users. XWiki P2P must be scalable so that it can handle larger numbers of users and work successfully. More precisely, scalability of P2P XWiki should be defined in terms of: number of users, data sizes, update workloads and query response times. Scalability is also related to extensibility of the system, which means the ability of installing new hardware and software resources easily.

Data Availability

To increase data availability in P2P systems (without relying on expensive special-purpose availability solution), the only viable solution is data replication. Therefore, XWiki P2P exploits data replication with the following advantages. First, replication improves XWiki data availability by removing single points of failure since objects are accessible from multiple peers. Second, it enhances XWiki P2P performance by reducing the communication overhead (objects can be located closer to their access points) and increasing throughput (multiple peers serve the same object simultaneously). Finally, replication improves the system scalability as it supports the growth of the system with acceptable response times. Since XWiki is essentially a collaborative application, optimistic replication is the best choice for XWiki P2P as it increases independent updating of the data. However, with optimistic replication, updating different replicas of the same data may cause replica divergence and conflicts, which should be reconciled.

Conflict Resolution

In client-server XWiki, users can share and update the same data using different sessions on the server. Thus, there can be conflicting updates of the same data page. To provide data consistency, XWiki implements a simple concept of conflict as follows. The owner of the page always has priority over any other user. Thus, in case of conflict, only the owner's updates are committed. If there is an update conflict between two users, none of them being the owner, then only the changes made by the last user are saved e.g. If user1 and user2 update a page. User1 save page before user2 and user2 save page after user1 then user2 updates will be save.
Client-server XWiki supports a page-level granularity for updating. This prevents multiple users to update different parts of the same page at the same time. Furthermore, conflicts detected at the

page level are solved by ignoring some changes, which could be integrated if a lower granularity were used. Thus, it makes much senses to increase the level of update granularity to line-level, a page being a sequence of lines. Conflict resolution must then be extended to deal with line-level granularity.

Mobility

Support of mobility is another important requirement for XWiki P2P. The objective is to provide mobile XWiki users, using lightweight devices such as PDA or smart phone, the ability to access XWiki data from other peers with possible disconnection and reconnection. This implies support for conflict resolution while reconnecting the network.

User Friendly Interface

Client-server XWiki provides an intuitive user-friendly interface, which made simple by the fact that the user understands the client-server abstraction and the server provides uniform access to all data. In XWiki P2P, with the data distributed and replicated at different peers, the same abstraction should be provided to users. In particular, the user interface should make it easy to deal with different peers to access data.

3.2.4 APPA (Atlas Peer-to-Peer Architecture)

APPA has a layered service-based architecture. Besides the traditional advantages of using services (encapsulation, reuse, portability, etc.), this enables APPA to be network-independent so it can be implemented over different structured (e.g. DHT) and super-peer P2P networks. The main reason for this choice is to be able to exploit rapid and continuing progress in P2P networks. Another reason is that it is unlikely that a single P2P network design will be able to address the specific requirements of many different applications. Obviously, different implementations will yield different trade-offs between performance, fault-tolerance, scalability, quality of service, etc. For instance, fault-tolerance can be higher in DHTs because no node is a single point of failure. On the other hand, through index servers, super-peer networks enable more efficient query processing. Furthermore, different P2P networks could be combined in order to exploit their relative advantages, e.g. DHT for key-based search and super-peer for more complex searching. Figure16 shows the APPA architecture, which is composed of three layers of services: P2P network services, basic services and advanced services.

P2P network services. This layer provides network independence with services that are common to different P2P networks:

- **Peer id assignment:** assigns a unique id to a peer using a specific method, e.g. a combination of super-peer id and counter in a super-peer network.
- **Peer linking:** links a peer to some other peers, e.g. by locating a zone in CAN.
- **Key-based storage and retrieval (KSR):**stores and retrieves a (key, object) pair in the P2P network, e.g. through hashing over all peers in DHT networks or using super-peers in super-peer networks. An important aspect of KSR is that it allows managing data using object semantic. Object semantic means that an object stored in the P2P network consists of a set of data attributes which can be accessed individually for read or write purposes.

This approach is appropriate for optimizing object access performance since we do not need to transfer the entire object through the network at each object access operation as the existing P2P networks use to do.

- **Key-based time stamping (KTS):**generates monotonically increasing timestamps which are used for ordering the events occurred in the P2P system.
- **Peer communication:**enables peers to exchange messages (i.e. service calls).

Basic services. This layer provides elementary services for the advanced services using the P2P network layer:

- **Persistent data management (PDM):**provides high availability for the (key, object) pairs which are stored in the P2P network.
- **Communication cost management:**estimates the communication costs for accessing a set of objects that are stored in the P2P network. These costs are computed based on latencies and transfer rates, and they are refreshed according to the dynamic connections and disconnections of nodes.
- **Group management:**allows peers to join an abstract group, become members of the group and send and receive membership notifications. This is similar to group communication systems [5, 4].

Advanced services. This layer provides advanced services for semantically rich data sharing including schema management, replication [13, 14, 15], query processing [1, 2], security, etc. using the basic services.

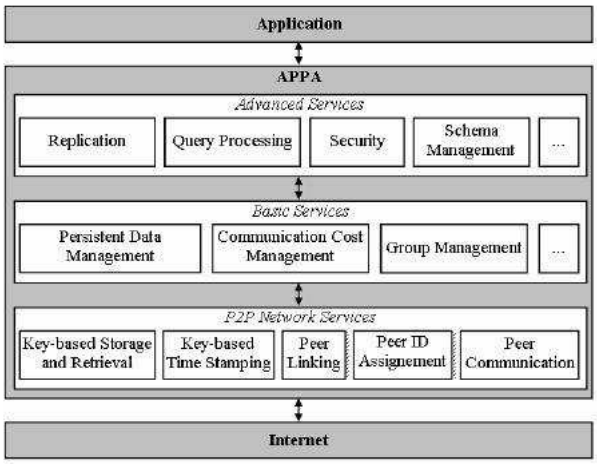


Figure 16: APPA Architecture

3.2.5 Architecture of XWiki using Semantic Store API

The Semantic Store API is an application programming interface that makes it easy for a P2P collaborative application to take advantage of data replication. By using this API, the application invokes the Semantic Store services while abstracts the Semantic Store architecture. Thus, the Semantic Store API works as a façade for the Semantic Store system, which receives service invocations, and then dispatches such invocations internally. In our case we use APPA as our Semantic Store. Figure 17 presents the use of Semantic Store API (APPA API) to integrate a

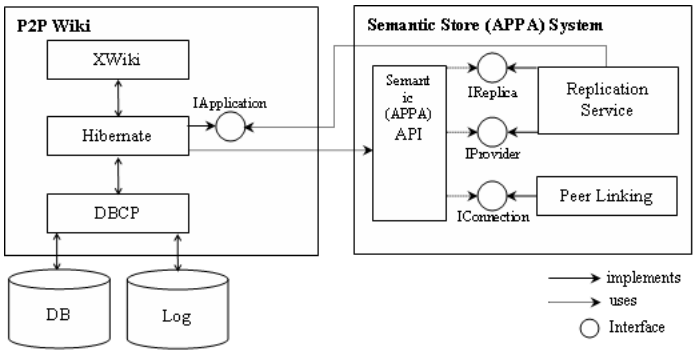


Figure 17: APPA API

Peer to Peer XWiki with the Semantic Store (APPA). For making XWiki P2P , We use a Semantic Store API (APPA API).The whole structure of XWiki will remain same except we will add three additional components, Log File, Invocation of Semantic Store API(APPA API),Implementation of IApplication Interface. Log File: The log file locally stores tentative update actions and constraints. For instance, if the wiki documents are built in XML, tentative update actions are the insertion, deletion, update, and move of XML elements. The Semantic Store API provide the following operations:

- **join()**: it connects an instance of the P2P Wiki to the P2P network that supports the collaboration; this operation triggers a replica synchronization that applies on the local replicas the global schedules produced while the peer was disconnected, if any exists. In addition, the replica synchronization requests that the P2P Wiki publishes the local log.
- **leave()**: it disconnects an instance of the P2P Wiki from the P2P network that supports the collaboration; this operation triggers a replica synchronization that applies on the local replicas the global schedules produced while the peer was connected, if any exists. In addition, the replica synchronization requests that the P2P Wiki publishes the local log.
- **Synchronize ()**: it performs replica synchronization on demand, which involves applying available global schedules and publishing the local log.
- **StoreActions (log)**: it stores into the P2P network the update actions present in the local log; this operation is part of the publication of local log that takes place at every connection, disconnection, and synchronization on demand.
- **StoreUserDefinedConstraints (cnt)**: it stores into the P2P network the user-defined constraints present in the local log; this operation is part of the publication of local log that takes place at every connection, disconnection, and synchronization on demand.
- **Start Reconciliation ()**: this operation launches the reconciliation of update actions already published but not yet reconciled. If the reconciliation is successfully started, a new global schedule sch is produced.

3.3 XWiki P2P Use Cases

In this section, we illustrate the use of XWiki P2P focusing on how the XWiki application communicates with APPA through the Semantic Store (SS) API. To show the use cases, we consider a group of three users, each being an XWiki peer. These users need to share a document with the following simple scenario. User 1 creates a document to be shared with users 2 and 3. After user

1 has saved the document for the first time, the document is transparently replicated at users 2 and 3 using APPA. Then, users 2 and 3 can independently work on the document and update it locally. All the local updates are captured through the SS API in the local action log at each peer. Furthermore, the SS API allows the action logs to be published to other peers through APPA. Upon reconciliation, the action logs are sent to a reconciler peer, which can be one of the user peers or another peer chosen by APPA. The reconciler peer applies the reconciliation algorithm to produce a global schedule (in a global log) which is then sent to the user peers. When receiving a global schedule through APPA, an XWiki peer applies the changes to the document and saves it in its local database.

In this section, we give three use cases, which illustrate this scenario: creation of a replicated document, parallel updating of the document, and reconciliation of parallel updates.

Creation of a Replicated Document

Consider user1 wants to create a document D on his XWiki, and wants to save it in the P2P network. He is the owner of the document D in his group. Figure 18 shows a simple scenario for replicating Document D in P2P network.

1. User1 creates a document D, to be replicated at peer2 and 3.
2. User1 updates D by adding one line L and saves D in the local database. A log file is created and captured details about document D by using the Semantic Store API. $D = LineL = Hello$
3. Semantic Store API makes easy for XWiki P2P application to take advantage of data replication. After D has been saved, the SS API is called by the XWiki client to replicate D on peer 2 and peer3 using APPA replication service.

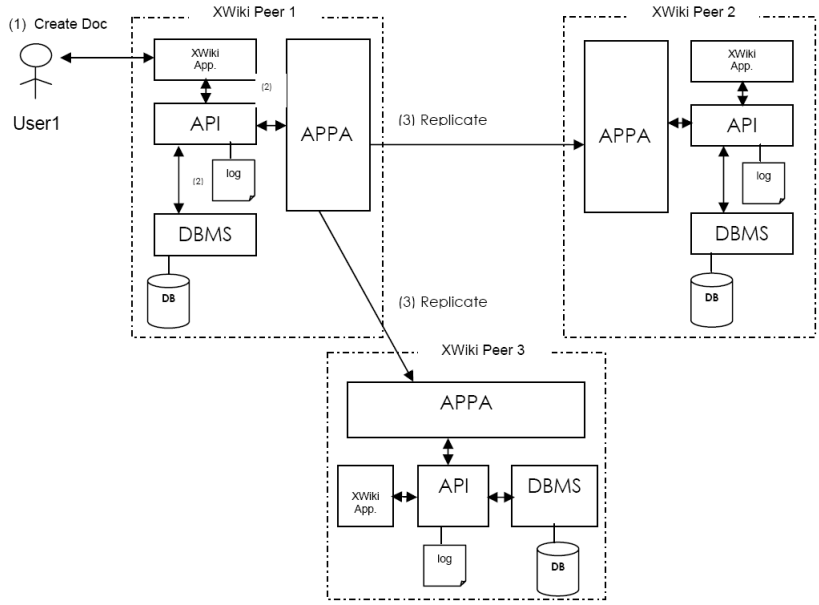


Figure 18: Creation of a replicated Document D

pseudo-code of Replicated Document

```
Replicate_Doc (PeerGroup , DocId)
Input:
  G: Peer Group;
  DocId:  Unique Identifier for a document D;
Begin:
  Doc= Get_Doc (DocId);   //Return Doc
  For Each  peer in G  do {
   Insert Data (Doc, p);  // Store data in peer2 and peer3
  }
End
```

Update of a Replicated Document

User 1 (the owner of D) and user 2 now work in parallel on D and perform updates. Figure 19 shows the corresponding use case.

1. User1 updates D by writing on line L1 (L1= ”Martin”) and saves the new state of D, D1, in the local database. On Peer1, the log file Log1 stores all update actions. i.e.

   ```
   { D1 = D+L1 }
        = Hello
          Martin
   ```

2. User2 updates D by writing on line L2 (L2= ”How are you?”) and saves the new state of D, D2, in the local database. On Peer2, the log file Log2 stores all update actions. i.e.

   ```
   { D2 = D+L2 }
        = Hello
          How are you?
   ```

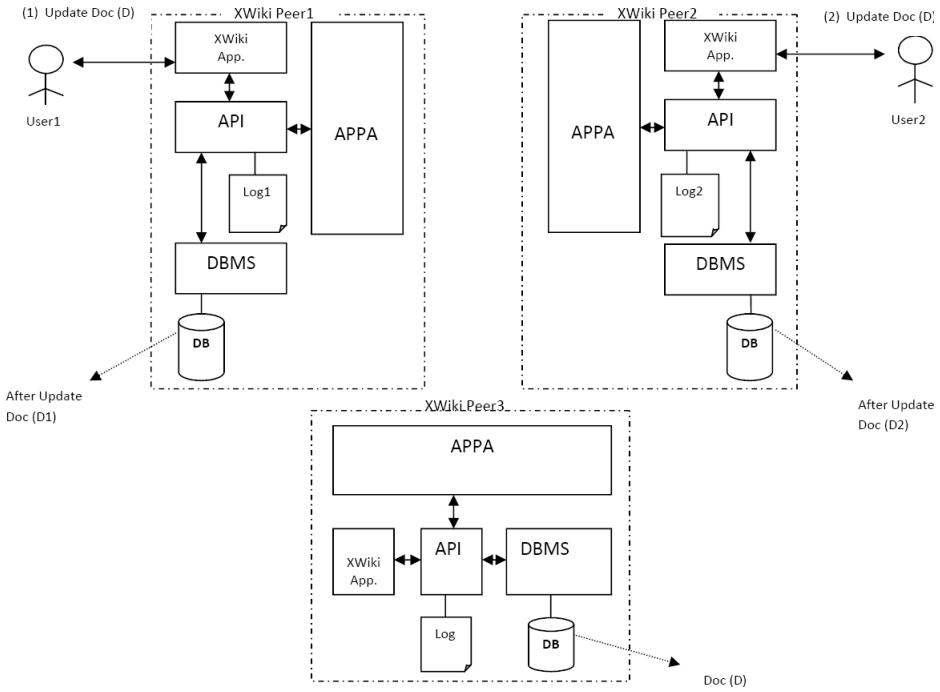


Figure 19: Update Document D

pseudo-code of Update Document

```
Update_Doc (Actions, Doc)
Input:
 Actions: Set of Actions (Insert, Delete, Update);
 Doc: Document;
Output:
Logfile: Log file contains actions
Begin:
 //Local Update
 Insert(Actions , Doc); // Update local document
 //Generate Log
 Log Listener (Actions):= Logfile;
 // called by XWiki peer where there is change in Document in order
 // to  store all tentative actions in log file
 Return (Logfile);
End;
```

Reconciliation

Peer3 is using as reconcile peer and apply the reconciliation service to log files and send back the global log contains schedule to peer1 and peer2. Figure 20 shows that Peer3 sends a request to peer1 and peer2 for reconciliation. In response, peer1 and peer2 send their local log files to peer3. Peer3, the reconciler peer, applies the reconciliation algorithm after receiving the log files from the other peers. Log reconciliation resolves conflicting updates and produces a global schedule that, when applied to all replicas, will lead them to a common, consistent state. After reconciliation document D become D3. i.e.

```
{ D3=D+D1+D2 }
    = Hello
      Martin
      How are you?
```

Figure 21 shows that after reconciliation process, peer3 generates global log named log3 contains global schedule, and send it to peer1 and peer2.Peer1 and peer2 apply the global schedule to their local replicas using their Semantic store API after extracting information from global log i.e. log3. Now the status of document becomes D3 on each peer i.e.

```
Hello
Martin
How are you?
```

pseudo-code of Reconciliation Document

```
Reconciliation_Doc (LogFile)
Input:
 LogFile: Log file from peer2 and peer3;
Output:
 Global_log: Log file generate after reconciliation;
Attribute:
 Peer Group [peer1, peer2,peer3]: set of peers;
```

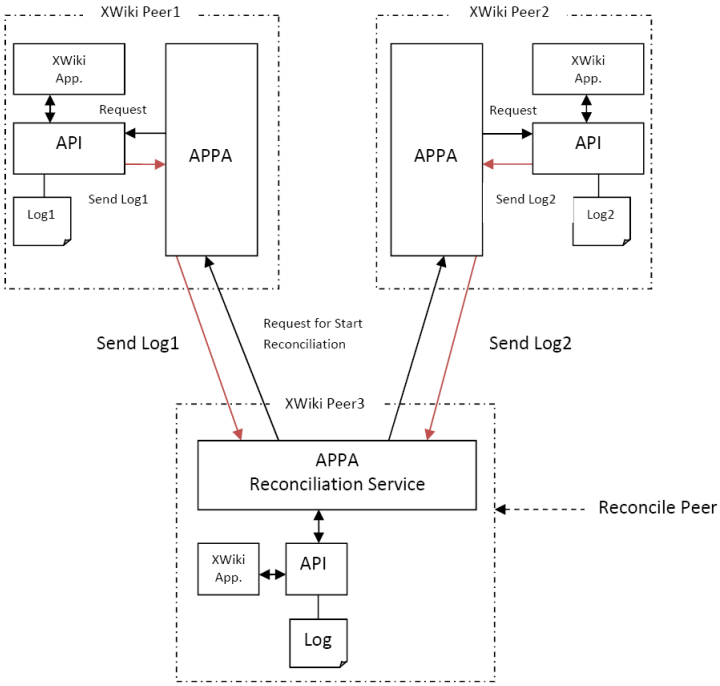


Figure 20: Reconciliation

```
Begin:
//Reconcile peer(peer3) send request to  peer1 and peer2 for publish their log.
 Request Publish Log();
 For peer in Peer Group do {
// peer1 and peer2 publish their log to  peer3 through APPA system.
  Publish Log (Logfile);
 }
 If all the logfile received then {
  Start Reconciliation():= global_Log;
// this operation launches the reconciliation of different log publish by
// peer1 and peer2 and return global log.
 }
// Send new log generate after reconciliation
 For Each peer P in Peer Group do {
  sendGlobal_Log (global_Log, p); // send Global log to  peer1 and peer2
 }
End
```

pseudo-code of Apply Global Log

```
Apply GlobalLog (GlobalLog)
Input:
 Global_Log;
Begin:
 When APPA receive GlobalLog from reconcilie peer do {
  //Semantic Store API extract information from GlobalLog
```

```
  SS_API_ExtractActions();
  //Update local database
  SS_API_ApplySch();
 }
End
```

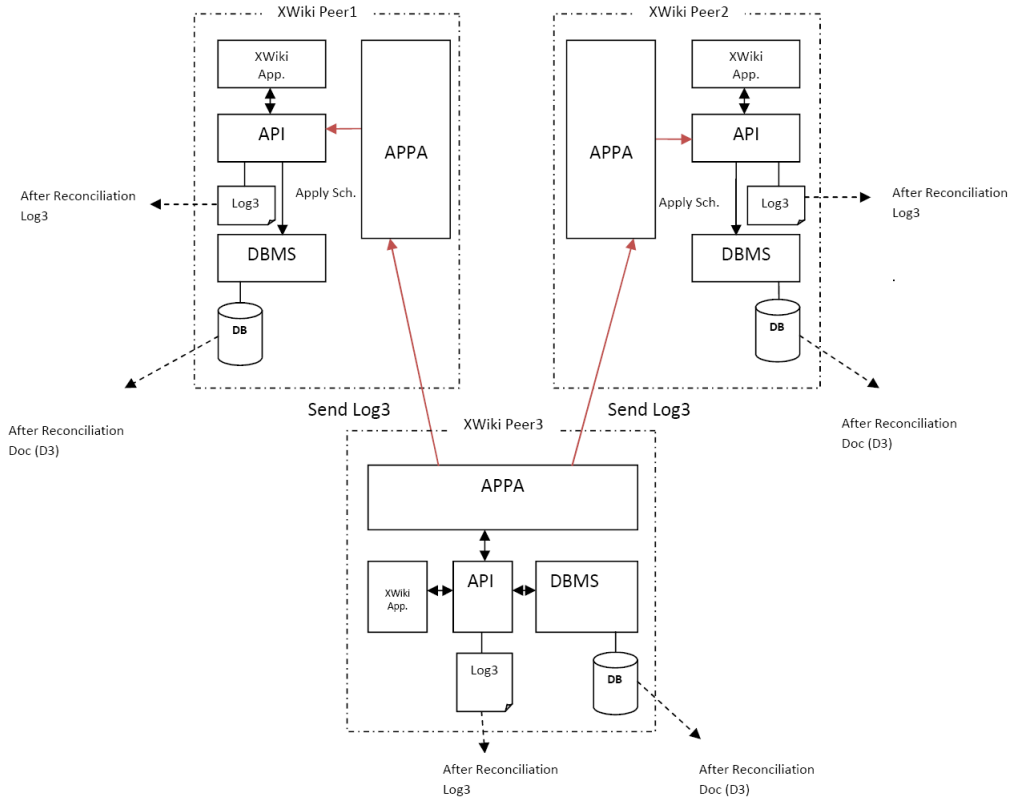


Figure 21: Reconciliation

References

[1] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Top-k query processing in the appa p2p system. In *In Proc.é of the Int. Conf. on High Performance Computing for Computational Science (VecPar),*, Rio de Janeiro, Brazil, july 2006.

[2] R. Akbarinia, E. Pacitti, and P. Valduriez. Reducing network traffic in unstructured p2p systems using top-k queries. In *Distributed and Parallel Databases 19(2-3)*, pages 67–86, may 2006.

[3] CALCONNECT. The calendaring and scheduling consortium, Last visite 05/05/2007. http://www.calconnect.org/index.shtml.

[4] M. Castro, M.B. Jones, A-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *In Proc. of the Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1510–1520, San Francisco, California, april 2003.

[5] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications a comprehensive study. In *ACM Computing Surveys 33(4)*, pages 427–469, CNAM, Paris, France, December 2001.

[6] DateDex. A calendar sharing directory, Last visite 05/05/2007. http://www.datedex.com/.

[7] F. Dawson and Stenerson D. Internet calendaring and scheduling core object specification - icalendar. Technical Report 2445, RFC, November 1998.

[8] W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative applications with bayou. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 119–128, New York, NY, USA, 1997. ACM Press.

[9] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 399–407, 1989.

[10] J. Ferrié, N. Vidot, and M. Cart. Concurrent undo operations in collaborative environments using operational transformation. In *Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 155–173, 2004.

[11] iCalShare. A calendar sharing directory, Last visite 05/05/2007. http://www.icalshare.com/.

[12] A-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of diverging replicas. In *In Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 210–218, August 2001.

[13] V. Martins, R. Akbarinia, E. Pacitti, and Patrick Valduriez. Reconciliation in the appa p2p system. In *IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society, Juillet 2006.

[14] V. Martins and E. Pacitti. Dynamic and distributed reconciliation in p2p-dht networks. In *In Proc. of the European Conf. on Parallel Computing (Euro-Par)*, pages 337–349, Dresden, Germany, September 2006.

[15] V. Martins, E. Pacitti, R. Jimenez-Peris, and P. Valduriez. Scalable and available reconciliation in p2p networks. In *. In Proc. of the Journées Bases de Données Avancées (BDA)*, Lille, France, October 2006.

[16] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *ACM SIGGROUP Int. Conf. on Supporting Group Work (GROUP)*, pages 212–220, 2003.

[17] G. Oster, P. Urso, P. Molli, and A. Imine. Réplicati optimiste dans les éditeurs collaboratifs répartis. In *Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR), ACM SIGOPS*, pages 155–173, 2005.

[18] Leysia Palen. Social, individual and technological issues for groupware calendar systems. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 17–24, New York, NY, USA, 1999. ACM Press.

[19] Leysia Ann Palen. *Calendars on the new frontier: challenges of groupware technology*. PhD thesis, University of California at Irvine, Irvine, CA, USA, 1998.

[20] C. Palmer and G. Cormack. Operation transforms for a distributed shared spreadsheet. In *ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 69–78, 1998.

[21] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *sosp*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS. http://www.parc.xerox.com/csl/projects/bayou/.

[22] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible update propagation for weakly consistent replication. In *ACM Symp. on Operating Systems Principles (SOSP)*, pages 288–301, 1997.

[23] Sushil K. Prasad, Anu G. Bourgeois, Erdogan Dogdu, Raj Sunderraman, Yi Pan, Sham Navathe, and Vijay Madisetti. Implementation of a calendar application based on syd coordination links. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 242.1, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Nuno Preguiça, Marc Shapiro, and J. Legatheaux Martins. SqlIceCube: Automatic semantics-based reconciliation for mobile databases. Technical Report TR-02-2003 DI-FCT-UNL, Universidade Nova de Lisboa, Dep. Informática, FCT, 2003. http://asc.di.fct.unl.pt/~nmp/papers/sqlice3-rep.pdf.

[25] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge, UK, May 2002. http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-2002-52.

[26] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth Int. Conf. on Coop. Info. Sys. (CoopIS)*, volume 2888 of *lncs*, pages 38–55, Catania, Sicily, Italy, nov 2003. springer. http://www-sor.inria.fr/~shapiro/papers/coopis-2003.pdf.

[27] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 38–55, 2003.

[28] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Int. Conf. on Principles of Distributed Systems (OPODIS)*, 2004.

[29] Marc Shapiro and Nishith Krishna. The three dimensions of data consistency. In *Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR)*, pages 54–58, CNAM, Paris, France, nov 2005. http://www-sor.inria.fr/~shapiro/papers/cdur2005.pdf.

[30] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 59–68, 1998.

[31] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention preservation in real-time cooperative editing systemss. In *ACM Transactions on Computer-Human Interaction*, pages 63–108, 1998.

[32] C. Sun, Y. Yang, Y. Zhang, and D. Chen. A consistency model and supporting schemes for real-time cooperative editing systems. In *Australian Computer Science Conference*, pages 582–591, January 1996.

[33] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Conf. on Comp.-Supported Cooperative Work (CSCW)*, page 59, Seattle WA, USA, November 1998. http://www.acm.org/pubs/articles/proceedings/cscw/289444/p59-sun/p59-sun.pdf.

[34] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM Symp. on Operating Systems Principles (SOSP)*, pages 172–183, 1995.

[35] N. Vidot, M. Cart, J. Ferrie, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 171–180, 2000.

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.).

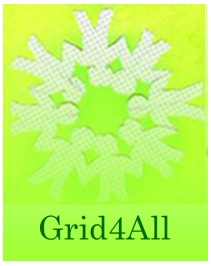
In the Grid4All Description of Work (DoW), and in the future yearly updates of the 18-months implementation plan, all deliverables listed in Section 7.7 have a specific dissemination level. This dissemination level shall be mentioned in the document (a specific section for this is included in the template,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



Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)

Thematic Priority 2: Information Society Technologies

Deliverable 3.1: Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities -- Chapter III

Due date of deliverable: 20 June 2007

Actual submission date: 20 June 2007

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

Organisation name of lead contractor for this deliverable: INRIA

Revision: Submitted 2007-06-20

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)	

Revision history			
Date	**Edited by**	**Status**	**Changes**
2007-03-16	Hamid Mizani, Vladimir Vlassov, Konstantin Popov	Draft	Initial draft of the chapter
2007-04-05	Hamid Mizani	Draft	First draft of the chapter 3 for deliverable; format it according to the deliverable template
2007-05-07	Hamid Mizani	Draft	Revised according to comments from Vladimir Vlassov
2007-05-15	Hamid Mizani, Vladimir Vlassov	Draft	Revised according to comments from Marc Shapiro

Table of Contents

List of Figures

List of Tables

Abbreviations used in this document

Abbreviation / acronym	Description
ACL	Access Control List
AFS	Andrew File System
CAS	Community Authorization Service
CFS	Cooperative File System
CHB	Content Hash Block
DFS	Distributed File Systems
DHT	Distributed Hash Table
FUSE	File system in User space
GFAL	Grid File Access Library
GSI	Grid Security Infrastructure
GT4	Globus Toolkit 4
GUID	Global Unique Identifier
LFN	Logical File Name
LRC	Local Replica Catalogue
MDDB	Meta Data Database
NFS	Network File System
OGSA-DAI	Open Grid Services Architecture-Data Access and Integration
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PERMIS	Privilege and Role Management Infrastructure Standards
POSIX	Portable Operating System Interface
RBAC	Role Based Access Control
RFT	Reliable File Transfer
RLI	Replica Location Index
RLS	Replica Location Service
RPC	Remote Procedure Call
SAML	Security Assertion Markup Language
SSO	Single Sing On
UFS	Unix File System
VFS	Virtual File System
VO	Virtual organization
VOFS	Virtual Organization File System
VOMS	Virtual Organization Membership Service
XCAML	eXtensible Access Control Markup Language

Grid4All list of participants

Role	Participant N°	Participant name	Participant 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 the Chapter III of Deliverable 3.1 ”Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities”, which comprises the following parts:

Chapter I	Semantic Store
Chapter II	Collaborative Applications
Chapter III	VO-aware File system
Appendix I	Telex application API

The following persons contributed to this chapter:

Vladimir Vlassov, KTH
Hamid Mizani, KTH
Konstantin Popov, SICS

1. Introduction

Data management is an important research and development topic in Grids because data are fundamental to all fields of society, science, business, and engineering. Data management in Grids deals with technical challenges and solutions aimed at providing integration, access, analysis and management of data distributed among multiple administrative domains. There are many data-intensive scientific applications that require mechanisms to transfer, replicate, discover, share, collect and analyze large amount of data from diverse geographically distributed data sources. A technical challenge for data-intensive business applications is to integrate, replicate and access data from different databases as well as to maintain database consistency in a large scale.

In general, data-oriented services in Grids can be divided in two major categories (1) data access and analysis, e.g. data mining and statistical analysis; (2) data transfer. In either case, a typical Grid application processes data from multiple heterogeneous data sources (e.g. databases, file systems) and sinks distributed among multiple administrative domains. An important issue in data management is integration of heterogeneous data resources in order to provide a unified data access and movement with a unified VO security mechanism.

Different applications have different demands for data-oriented services depending on what kind of data a particular application is processing and how the data are to be processed. Some applications require a conventional POSIX API to access files; whereas other applications require high-level services with more sophisticated API to perform analysis (e.g. data mining) of data stored in databases or XML files. The lower level data access services, provided to application developers, resemble POSIX data access. The higher level data-oriented services provide ability to query databases or XML files, to query RDF and RDFS data, to manipulate data objects with specific semantics and a specific API, e.g. multilogs. The OGSA-DAI [] software defines the state of the art for database access. As indicated in [GT4 white paper], many Grid-related application-oriented EU projects are using OGSA-DAI in different Grid applications which are in focus of the projects. However, many scientific applications (including legacy applications) are using files; therefore several large-scale (global) file systems have been proposed and implemented within Grid projects, e.g. Grid File Access Library (GFAL) offered and used in the LHC Computing Grid (LCG) as a part of the gLite Grid middleware, which is currently used and developed within the EGEE project.

Considering database access and high-level services is out of the scope of this document that is focused on Grid-wide distributed file systems, specifically, Virtual Organization file systems (VOFS), and use of P2P technologies in file systems.

In Grid4All project, we consider ad-hoc Grids, which contain different type of resources, e.g. computers, storage, networks, etc. These resources are voluntarily provided by users and organizations, e.g. schools and SMEs, forming VOs. VOs might have short lifetime, they might overlap each other and can be highly dynamic. This dynamicity applies to both resources and VO members. In highly dynamic VOs, members as well as resources can join and leave very frequently. Users and resources might simultaneously participate in multiple VOs.

The core services of the Grid infrastructure (such as execution management, monitoring and discovery, data management, information services and security) should be deployed on the resources which are rather stable and (highly) available. On the other hand, the infrastructure should be able to tolerate the churn and to use the resources provided by ordinary users as well. The ad-hoc Grid infrastructure should also be able to react on the VO evolution (changes in the number of resources and VO members), i.e. to scale and to adapt quality of services to changes in the VO. In order to tolerate churn, the ad-hoc Grid infrastructure should provide efficient replication of services, data and metadata as well as efficient mechanism for service hand-over especially for critical services. Furthermore, this should be done automatically wherever possible, i.e. an ad-hoc Grid should be self-managing. This can be achieve by providing a control interfaces and a control logic on services (Grid components).

File (data) transfer and replica management can be either components of the Grid file system or separate services that compliment the file system with corresponding functionalities. Major requirements to a data transfer service are high reliability and performance. One of the major requirements to a replica management system is the ability to maintain consistency of mutable replicas.

VOFS should be able to aggregate different file systems including DFS (Distributed File Systems) to provide data sharing between VOs. A DFS provides access to remotely located files to the clients in a distributed system. DFS can be built on top of VBS (Virtual Block Store) which acts as a storage manager layer and provides storage to other systems (like DFS) to store files. VOFS should also provide support for disconnect operations which can be implemented using local caching and a reconciliation mechanism. VOFS should support file replication for the sake of performance, robustness and availability. In order to obtain storage resource for replicas and VOFS services such as Metadata servers (file catalogs), Mount services (for persistency), VOFS will interact with a resource (VO) management system to be developed in Grid4All Work Package 2.

In this document first we describe the state of the art in Grid file systems. We also study the state of the art in data transfer and replica management in Grids, because those services and a VO file system are closely related to each other. For example, a Grid file system can use file replication in order to achieve higher performance and availability; whereas a replica management sub-system uses a file (data) transfer service to move replicas and to maintain replica consistency. Then we specify requirements and design issues for VOFS, and based on them describe a possible design of the Grid4All VOFS. In next section we focus on the VO-awareness properties of VOFS and security issues, and we explain the mechanisms we plan to use. Then we define a usage scenario to show how VOFS works in a simple but real situation. Finally we describe the implementation details of a VOFS prototype.

2. State of the art in Distributed and Grid File Systems

This section describes state of the art in distributed file systems (DFS thereafter), serverless/P2P DFSs, and file systems developed specifically for Grid environments. We study requirements, features and solutions in DFSs (architectures, protocols, caching, replication, consistency, and security mechanisms) that can contribute to the VOFS design.

First, we give an overview of distributed file systems that assume the client-server model. DFS can operate either within one administrative domain (e.g. NFS: Network File System) or across multiple administrative domains using the same security mechanism (e.g. AFS: Andrew File System). Each administrative domain has own users, user groups, security mechanism and administrators. Servers in DFS can be either centralized (NFS) or distributed (AFS, Sprite, Frangipani). DFS can support replication of data over multiple servers (Ficus, Coda) and aggressive caching supporting high availability, throughput, and/or disconnected operation.

Next, we consider serverless/P2P-based files systems. Such DFSs do not distinguish between server and client nodes, and have decentralized control and storage (xFS, Farsite). They aim at scalability, throughput, and/or robustness and security. Besides such DFSs that provide, by definition, a POSIX-like interface for file access, there are also P2P-base data storage systems that have different client requirements and interfaces (such as support for versioning) but aim at the same properties of data storage (Pond/OceanStore, LegionFS/AvakiFS). Some designs employ a P2P-overlay and/or DHT layer that is used for storing data and directory blocks, as well as accessing peer nodes (Pond/OceanStore, Keso).

Finally, this chapter presents an overview of several file systems specially developed for Grids. While DFSs design can serve as a starting point for Grid/VO-aware file systems, there are Grid/VO-specific issues like compatibility with the VO life cycle model, security and trust, resource- and client- heterogeneity, and necessary integration with the VO resource management.

2.1 Distributed File Systems

In this section, we consider a few most commonly used distributed file systems (DFS) that operate using the same network file protocol with the same security mechanism. A DFS provides access to remote (distributed) files via a local file system. A typical DFS has a client-server architecture that includes a set of remote servers that provide access to remotely located files for clients accessing the files, and an access protocol that is typically RPC-based. A DFS may also contain additional servers for authentication and authorization like in AFS (Andrew File Systems). The server and the client can be in either in the kernel space or user-level processes. Examples of ordinary DFS are NFS (Network File System), AFS (Andrew File System), Windows DFS. Most of distributed file systems like those mentioned above, operate within one or multiple (e.g. AFS) administrative domains but using the same security mechanism (e.g. Kerberos tickets). For example, in AFS, each cell is a separate administrative domain with its own accounting system, i.e. own users, users groups and administration.

2.1.1 NFS: (Sun) Network File System

NFS (Network File System)[26] was first introduced by Sun Microsystems in 1985. Since then, the NFS protocol became the de facto standard network file protocol for Linux, Unix, SunOS, and Solaris operating systems. Even though NFS is particularly common on UNIX-based systems, NFS implementations are available on almost any platforms including Windows. However, only when used in a UNIX-based system, NFS closely resembles the behaviour of a client's local file system. The most commonly used version of the NFS currently is NFSv3 and the latest NFSv4. The latter addresses some weaknesses in earlier NFS versions such as ACL, security, and file system namespace, etc. However, NFSv4 is not used as extensively as NFSv3.

The NFS model is a remote file service that offers clients transparent access to remote file systems managed by remote servers. Clients are actually unaware of the actual location of files, which is also known as name transparency. A client can mount and access a remote file system exported by an NFS server through the NFS mount point on the client's local file system. The access is transparent to the client application. Unlike many client server communications, NFS uses remote procedure calls (RPC) for communications. The client connects to a known port on the server and then uses particulars of the protocol to request specific action. NFS is a stateless protocol. This means that the file server stores no per-client information, and there is no a notion of an NFS "connection" established between a client and a server. Both, NFS client and NFS server, maintain caches of memory blocks for efficient file access. The server-side cache is "write-through" using time stamps on file open; whereas the client-side cache is "write-back" with delayed writes on file close and sync. The "write-through" caching policy at the server side is a consequence of server statelessness, and it severely impacts the 'write' NFS performance. NFS supports open-close weak consistency.

2.1.2 The Sprite Network Operating System

The Sprite network operating system [27] aimed at true network transparency for applications running on workstations interconnected by a local-area network. The Sprite OS included an efficient distributed file system that provided network transparency for applications.

A Sprite file system consists out of domains on different server machines that together represent a single hierarchical directory structure shared by all workstations. The Sprite file system offers name transparency. The Sprite file system utilizes all available workstation memory for caching at both the client and the server sides, yielding far superior performance compared to e.g. NFS. Sprite uses a simple cache consistency mechanism that guarantees that applications running on different workstations always observe the most up-to-date version of data. The consistency mechanism optimizes the case when a file open for update is not open at any other workstation. Locating a workstation responsible for a particular file is facilitated by so-called prefix tables which map path prefixes corresponding to domains onto workstations currently responsible for those domains. Prefix tables are updated automatically using a LAN broadcast, which greatly simplifies management of a Sprite installation site while providing satisfactory performance. The statefullness of the Sprite file system protocol actually provided a solution for application-transparent recovery after server crashes [28]: clients cache and provide the necessary information to a server after its reboot. This achieves the level of robustness provided by NFS but with far better performance, albeit at the expense of the protocol complexity and certain memory requirements.

The Sprite file system paved the way for more efficient prototypes such as log-based striped distributed file systems Zebra [36].

2.1.3 AFS: Andrew File System

The Andrew File System (AFS)[29] is a distributed networked file system developed by Carnegie Mellon University as part of the Andrew Project and now funded by IBM within the OpenAFS project [13]. The main AFS design principles are that (a) whenever possible, perform an operation on a client workstation thus relieving servers, (b) cache data whenever possible, (c) minimize system wide knowledge, (d) trust the fewest entities possible, and (e) batch operations whenever possible.

AFS distinguish local and shared files. The latter are stored on AFS servers and cached on the client's disk. Shared files are named in the AFS uses global name space with the root directory called /afs that is (can be) mount to a local file system to the /afs mount point. The AFS file system is formed of a collection of basic administrative units called cells. In the AFS directory structure, cells are located under /afs. A cell, e.g. "it.kth.se", constitutes a separate administrative domain of authority; it has its own list of users, groups, and system administrators. Each cell is made up of volumes. A volume is a named collection of files and directories that are grouped together as a data unit (e.g. a user's home directory) that can be moved from one server to another, backed up, replicated or destroyed..Volumes are assigned volume quotas in K-byte blocks.

Like many other distributed file systems, AFS uses client-side caching in order to reduce network traffic when accessing remote files. A special client cache manager is used to maintain a file cache on the client's computer. The cache manager is invoked by the local operating system when the latter misses in the file cache or the cached copy needs to be written back to the AFS server on file close. On open, the AFS server issues a "callback" promise to notify the client if the file is updated. On close, the cache manager sends the file to the server, which, in its turn, invalidates callback promises on copies (if any). The callback mechanism, thus, requires the AFS server to maintain information on clients holding cached copies of the file. AFS also has provides replication by using a single read/write copy backed by one or more read-only duplicates. This paradigm also permits users to read data from a duplicate copy, while reserving the read/write master for operations requiring write access which improves availability. AFS clients are also provided with an automatic fail-over capability, allowing them to detect the loss of a server and connect to another machine with no user intervention required. Like NFS, the AFS file system supports a weak consistency model with the open-to-close (download-to-upload) semantics.

The security mechanism (authentication and authorization) in AFS is based on and Kerberos and ACLs (Access Control Lists) set on the directory level. An ACL can contain up to 20 users and/or group entries with specified granted rights, e.g. read, write, delete, etc. AFS assumes integrity of servers while client workstations are considered untrustworthy.

2.1.4 The Coda File System

The Coda file system[29][30] is a descendant of AFS. Coda strives for high data availability, allowing users to work with data even when some or even all file servers are temporarily not accessible. Coda retains AFS' caching schema with server callbacks for maintaining cache coherence, dynamic discovery of file locations and caching of that information, and using token-based authentication and end-to-end encryption as the basis for file system security. Coda relies on two mechanisms to achieve its failure resilience goal: server replication and disconnected operation mode. Update conflicts between different servers are resolved as soon as possible. In normal operation, when all servers are available, Coda clients read data from a singe server, check data status from all servers, and update all servers. Disconnected operation relies on aggressive caching of data. During disconnected operation, Coda client software acts as a server replica, and latter on is converted back to cache management function. Coda attempts automatic conflict resolution on the directory structure, and relies on manual conflict resolution on files. The latter is assisted by certain CODA tools that make conflicting versions of data to be easily accessible to the user who attempts to resolve conflicts.

2.1.5 The Ficus Replicated File System

The Ficus replicated file system[32] design provides for highly-available file storage built from "off-the-shelf" native file systems available on Unix platforms, and NFS used to access remote native file systems. Ficus allows operation under the network partitioning, though it requires manual replica reconciliation should conflicting updates emerge. Conflicting updates to directories are repaired automatically.

Ficus has a layered architecture, each layer using the vNode interface. The vNode interface is used within most flavours of Unix (including Linux) for managing file systems, abstracting file system-specific details from the rest of the operating system. In the Ficus architecture, there is a "Ficus Physical" layer between the native file system layer and NFS, and a "Ficus Logical" layer between NFS and the OS system calls available to file system clients. The logical layer manages the file name space and file replicas, while the physical layer implements file replicas.

A Ficus file system consists out of volumes that are grafted (mounted) together. A volume is a collection of files that are managed together. A mount point is represented by a special entry that contains the volume identifier and a list of volume replicas. Such entries are used during path translation to cross the boundaries between volumes. Replication and consistency of such entries is managed using the standard Ficus mechanisms for data replication.

2.1.6 Frangipani : A Scalable Distributed File System

The Frangipani DFS[31] aims at scalability, high availability and minimal administration by humans. The Frangipani design attempts to address these challenges by utilizing a two-layer structure, where the lower layer is a distributed storage service providing a single shared virtual disk with aforementioned properties and the upper layer is a file system that runs on top of the virtual disk storage. Upper layer servers do not communicate with each other, but only with the lower layer virtual disk storage (and the lock server). Frangipani assumes a cluster-like environment which has common administration and where communication is secure. More disk space can be added to a Frangipani DFS through the lower level, without affecting configuration of any of the existing Frangipani servers. Also, it is possible to take a backup of a file system while it is being used.

Frangipani's file system layout is reminiscent to original Unix file systems like Sun's UFS. Such a simple layout is allowed by the sparse storage provided by the virtual disc layer. Crash recovery is facilitated by the write-ahead redo log of metadata (user data is not logged). A fine-grained locking schema facilitates scalability of the file system.

2.2 Serverless/Peer-to-Peer File Systems

P2P is decentralized networking paradigm in which peers communicate and work collaboratively to share resources, provide and consume services.

The convergence of peer-to-peer (P2P) and Grid systems has been recognized as a natural evolution to bridge the gap between the two, merging their respective strengths and resolving their respective weaknesses. Both P2P and Grid systems are build to share resources and to provide access to shared resource. However, in contrast to Grids, a typical P2P system has low guarantees and low trust.

Traditionally, P2P techniques are used for building file sharing systems such as Gnutella, Kazaa, Freenet as well as content delivery networks (CDN) such as Akamai. A P2P file sharing system is formed of peers making their files available for each other to download. There exist a large number of file sharing systems and CDNs.

There exist three types of P2P systems: unstructured system with random topologies, structured system with regular topologies (e.g. exponential P2P networks) and hybrid. In addition to traditional overlay network functionality, structured P2P systems such as Chord, Pastry, Tapestry, CAN, Tulip, and DKS, provide the DHT (Distributed Hash Table) functionality. DHT allows storing and retrieving different kind of data objects identified by keys. In a Grid system, DHT can used for a decentralized lookup (index) service, for example, to keep track of Grid resources and services, to store and retrieve RDF triples of semantic information, and, in particular, to store and retrieve files or file blocks.

A number of recent P2P file systems [CFS, Ivy, Pastis, Keso, MyriadStore] have been designed by providing distributed file system functionalities on top of DHTs (Distributed Hash Tables) offered by the structured P2P overlay. A global P2P DFS is built over the set of local file storage systems of individual peers. Like in ordinary file system, inodes and contents of files and directories are stored in fixed-size blocks which are put into the underlying DHT using public keys and content hashes as block identifiers. Inheriting the self-organizing properties of the overlay, these file systems are resilient, robust, and scalable.

2.2.1 xFS – A Serverless Network File System

The xFS[35] prototype demonstrated the feasibility of the "serverless network file system" approach. In xFS, every node can act as both client and a server, and any server node can store, control and cache any block of data, giving good availability. This approaches uses the location-independence principle, which, combined with caching and fast local area networks, gives better performance and scalability than traditional file

systems. Designs like xFS became timely because of (a) massive introduction of high-performance switched local area networks, and (b) new applications such as multimedia, process migration and parallel processing put higher requirements on file systems than can be handled cost-efficiently using client-server approaches.

xFS distributes control and storage evenly over all participating server nodes, it implements a software RAID using log-based network striping similar to Zebra[36], and it uses cooperative caching taking advantage of available memory and network bandwidth. The primary limitation of the approach is that it assumes trust between OS kernels running xFS server nodes. Also, there can be scalability problems in large-scale environments as xFS server nodes share certain globally replicated data structures.

The xFS's philosophy is to allow to store any information, both data and all kinds of metadata, anywhere. xFS maps file names to file indices, which, in turn, are mapped to disk log addresses of file's metadata (location of data blocks in the file). Log segments are partitioned among subsets of server nodes. Ultimately, a server node can retrieve any log block in the system. Additionally, xFS performs aggressive caching, and xFS server nodes prefer to request a data block from a peer server node than from a disk. xFS also features support for distributed cleaning of logs and crash recovery.

2.2.2 CFS: Cooperative File System

CFS [1] is a decentralized read-only DFS based upon the Chord P2P system and the DHash distributed block storage that is a DHT layer on top of Chord. DHash provides the block storage, caching and replication using Chord as a routing infrastructure. Files and directories are stored and retrieved as blocks in the DHT using cryptographic hashes of block contents as keys. CFS allows storing multiple file trees in the same DHT with one publisher (owner) per a tree. Each tree is immutable. In CFS, a publisher (an owner) of a tree stores all the data in content hash blocks using content hashes as identifiers; then she creates the root-block, signs it using her private key and stores it in to DHT using her public key as the block identifier. To improve the system performance, CFS uses virtual servers for load balancing.

2.2.3 IVY: A Read/Write P2P File System

Ivy is a read/write P2P FS that provides NFS-like semantics if the underlying P2P network is fully populated. The Ivy file system was developed by the same group that developed CFS and Chord, and, like CFS, Ivy is based upon Chord and the DHash distributed block storage. Ivy supports the open-to-close consistency and uses logs to support concurrent writes. It is intended for a small group of participants, in which each of the participants keeps her log. The log is split into blocks stored in DHT. The log-head is identified by an owner's private key and is signed by the corresponding private key. The reminder of the log entries are content hash blocks. Ivy uses views and snapshots to maintain consistency of multiple logs. A view is a set of logs that a number of participants have agreed to trust. The view comprises the file system. It has a view-block that contains all the log heads of trusted participants. Each node pf the system keeps private snapshot with the current state of the FS. This allows to avoid traversing the all snapshots. Snapshots are stored in DHash using content hashing. A snapshot is updated with log entries newer than the last snapshot.

2.2.4 PASTIS

Pastis [49] is a read-write file system that can scale to large numbers of nodes and users. It uses a modified version of the Past DHT based on Pastry [50] to store file system data. Pastis stores its data in data structures similar to the UNIX file system (UFS). The metadata of a file, similar to a UFS inode, is stored in mutable blocks which we call User Certificate Blocks (UCBs). In order to guarantee data authenticity, each UCB is digitally signed by the writer before it is inserted into the DHT. File and directory contents are stored in fixed-size immutable blocks called Content-Hash Blocks (CHBs). The key of a CHB is obtained from the hash of its contents, which makes the block self-certifying. The keys of a file's CHBs are stored in the file's inode block pointer table.

Pastis uses close-to-open and a variant of read-your-write consistency models. The close-to-open model is implemented by retrieving the latest inode from network when the file is opened and keeping a cached copy until the file is closed. Any following read requests are satisfied using the cached inode. New CHBs are also buffered locally instead of being inserted immediately into the DHT. Finally, when the file is closed all cached data are flushed to the network and removed from the local buffer. Since the immutable data blocks (CHBs) that store the contents of each different version of a given file (a new version appears each time the file is closed) are never removed from the network, a complex garbage collection mechanism would have to be employed to safely remove unused immutable block from the DHT. The read-your-writes model guarantees that when an application opens a file, the version of the file that it reads is not older than the version it previously wrote. Once the file is opened, file updates are performed as in the close-to-open model.

Pastis ensures write access control and data integrity through the use of standard cryptographic techniques and ACL certificates. Pastis does not currently provide read access control, but users may encrypt files' contents to ensure data confidentiality if needed.

2.2.5 KESO

KESO is a read/write file system built on top of the DKS P2P middleware. KESO supports file replication, versioning, access control using PKI, data privacy using encryption. The main goal of Keso is to use of spare resources and to avoid storing unnecessarily redundant data. It is a versioning file system, which means that old versions of files are kept in the file system after then have been changed. It provides a reliable file system on top of untrusted components. Security assumptions are based on minimizing trust, trust users and not computers, and finally correctness of the received data should be verifiable.

Keso splits files into same size blocks and encrypt them when it wants to save a file. These datablocks are then referenced from an inode. An inode consists of a blocklist of the datablocks of the file and the related metadata to the file. The inode also contains the keys needed for decryption of the datablocks.

The metadata contains data about the file itself, such as the time this version of the file was created, the list of permissions to use when users tries to access the file and who created it.

Directories provide a mapping between names and inodes. A directory contains a set of metadata, a directory lookup table and a signature. The directory lookup table is a list of entries which each contains a name and a list of file change entries. Subdirectories are treated in the same way as files. Each entry consists of an operation, the inode this operation relates to, and the user which has performed the operation. Operation can be one of either Create or Delete, where Create means that a new file version has been created and Delete means that the file has been deleted.

2.2.6 Farsite

Farsite[2], developed at Microsoft Research, Redmond, is a P2P (serverless) DFS that aims at harnessing available resources in large sets of insecure and unreliable machines in order to provide reliable and secure file systems. Farsite mimics the semantics of NTFS, the native FS of Windows NT.

In Farsite, a number of hosts dynamically form a directory group to manage a virtual namespace root. The hosts are responsible for metadata of that part of the FS. The directory group of hosts provides a group service on the metadata and maintains consistency of metadata using a variation of the Byzantine agreement protocol. Each file is replicated on a number of hosts (not necessarily those of the directory group) in order to distribute system load. File data is opaque to the file system; Farsite uses encryption and one way hashing provide for privacy and data integrity. A directory entry for a file contains a cryptographic hash of the file contents and a list of hosts where the file is stored. A client can retrieve the file data from any of the hosts and use the hash to verify the contents. The file content can be also encrypted such that only authorized users can read it. Since clients can verify data integrity, hosts keeping file data do not have to form groups running a Byzantine agreement protocol, which reduces the number of hosts needed to store

file data, as well as improves performance. Furthermore, file data integrity/privacy implementation allows incremental updates of file content without recomputing hashes resp. encrypting entire files after a modification. File directories are also encrypted so that hosts forming directory groups cannot read file names. The security mechanism in Farsite is based on public key cryptography, a certification a and ACLs.

Client's updates to a directory group's metadata are logged (similar to e.g. Coda) and propagated to the directory group periodically or when the client's lease expires or is revoken/updated. The Farsite's lease mechanism allows to control consistency of files under concurrent access. There are four classes of leases: content leases (that govern access to files), name leases (that govern access to directory structure), mode leases (that specify what a client is allowed to do with a data object) and access leases (that specify what other clients are allowed to do with the file concurrently with the client that obtained a given access lease).

2.2.7 OceanStore and its Pond Prototype

OceanStore [37][38] is an ".. internet-scale, persistent data store designed for incremental scalability, secure sharing, and long-term durability"[37]. The OceanStore architecture is two-layered: the upper layer comprises a set of tightly-knit set of powerful hosts that serialize changes and archive results. The lower layer encompasses hosts which provide storage for the system. The unit of storage in OceanStore is a data object. OceanStore does not explicitly deal with directory structures, and does not provide a POSIX-like interface. OceanStore provides for name transparency, well-defined data privacy, integrity and consistency models, OceanStore assumes that (a) the infrastructure is untrusted except in aggregate, and (b) the infrastructure is volatile.

Every OceanStore data object is represented by a sequence of read-only versions each of which is a B-tree of [read-only] blocks. Blocks are identified by secure hashes of their content. The same block can appear in multiple versions of the same data object (or different objects). An update is a set of actions each protected by a predicate. This flexible schema was inspired by the Bayou architecture which is ".. a platform of replicated, highly-available, variable-consistency, mobile databases on which to build collaborative applications"[39][40]. OceanStore resources, in particular data blocks, have globally unique identifiers and are stored in the Tapestry overlay network[41]. Each data object has a primary replica which manages consistency of object updates. Primary replicas are implemented as a set of hosts called the inner ring that run a computationally efficient version of the Byzantine agreement protocol. Archival storage of data object blocks is performed using the erasure coding scheme[42] that allows to reconstruct a data block from an arbitrary subset of certain minimal size of fragments into which the data block has been encoded. Since re-computing a data block from its erasure codes is expensive, OceanStore employs aggressive caching of data blocks.

2.3 Grid File Systems

Several Grid (global) file systems have been proposed and implemented that allow to provide their clients a global file system abstraction with logical file names, symbolic links, and POSIX-like APIs. Such file systems allow to support existing clients and legacy applications in Grids without major modifications, as well as developing new Grid applications using convenient file APIs familiar to users.

In this section we present several Grid (global) file systems, such as the gLite LCG File Catalogues and the Grid File Access Library (GFAL), AliEnFS, which define the state of the art in Grid file systems.

The major difference between an ordinary distributed file system and a Grid file system and is the former provides access to remotely located files using one protocol and one security mechanism (authentication, authorization and accounting), whereas the latter allows aggregating heterogeneous native file systems from multiple administrative domains in a global file system in which files are accessed and transferred using different standard and custom protocols such as NFS, HTTP, FTP, etc and POSIX-like APIs.

There are many similarities in designs and implementation approaches in different proposed Grid file systems. Most (if not all) of the proposed Grid file systems provide a global hierarchical name space in which files are identified by their Logical Files Names (LFNs). Typically, each VO has its own name space. Mappings of logical file names to physical names (including local information and access/transfer protocols) are typically stored in file catalogues, which, typically, use backend databases. Typically, file catalogs also maintain information on replicas, unless there is a separate replica location service or a replica catalog.

A grid file system can be mounted to local file systems of clients in order to allow applications to seamlessly access the grid file systems. A typical architecture of a Grid file system is client-server with P2P-like interactions between servers. Like in an ordinary homogeneous distributed file system, a Grid file system uses caching of files or parts of files on client sites in order to reduce network traffic and to achieve high-performance file access. Most of existing Grid file systems use replication to improve performance and robustness.

Most of file systems use loopback adapters or user-space file system solutions, e.g. LUFS (Linux Userland File System) [11] and FUSE (Filesystem in Userspace) [12] in order to avoid modification of local operating systems of participating computers. Such solutions allows to provide a plug-and-play mechanism to aggregate file systems of different types with different protocols as well as to provide VO-level security mechanism in the Grid file system without the need to modify the operating system.

2.3.1 The gLite File System: LCG File Catalogues

[3][4] The gLite Grid middleware used in the LHC Computing Grid (LCG) of the EGEE project, includes the following three main service groups that relate to data and file access:

- Storage Element (SE) that provides uniform access and services to large storage spaces. Each site includes at least one SE.
- Catalog services that, in particular, include LCG File catalogs (LFC). LFC is a file catalog service that keeps track of the location of copies (replicas) of files and relevant metadata (e.g. checksums and file sizes). It uses either Oracle or mySQL as a backend databases.
- Data movement services that allow for efficient managed data transfers between SEs.

All the data management services act on single files or collections of files. In overall, the EGEE data services present to clients a global file system abstraction with logical file names, symbolic links and a POSIX-like API. A client can browse and navigate this virtual file system by listing files, changing directories, etc. The files can be accessed via the Storage Element (SE) responsible for the files. The access to the files is controlled by Access Control Lists (ACL).

gLite provide several naming schemas to identify files and their replicas. Different file identifiers mentioned below are used by different data management services and client applications to access, to keep track of the files or/and to move files.

In gLite, a file can be identified by a human readable mutable Logical File Name (LFN) in a global hierarchical name space; and by immutable Global Unique Identifier (GUID). The Grid data management maintains logical-to-physical file name mappings in a scalable manner. A client application uses LFNs in a global hierarchical LFN namespace like in an ordinary file system, e.g. UNIX. Each Virtual Organisation can have its own namespace. In contrast to LFNs, which are mutable (files can be renamed by the user), GUIDs are immutable. The logical namespace also provides the concept of symbolic links that point to LFNs.

A replica (an instance) of the file can be identified by its logical Site URL (SURL) with the srm (Storage Resource Manager) protocol; and by a Storage URL (StURL) that is the actual file name inside the storage system. The SURL consists of two parts: the SRM end point and the LFN. An instance of the file (a replica)

can be also identified by its Transport URL (TURL) that specifies a standard transport protocol, e.g. HTTP or FTP, used to transfer the file and the file location (host and port).

gLite Storage: gLite provides several storage systems which are available to users via a unified SRM (Storage Resource Manager) interface. In addition on the SRM interface, the storage systems provide APIs for POSIX-like file access and file transfer.

Grid File Access Library (GFAL): To hide differences in POSIX-like APIs of the different storage systems, the gLite Grid middleware provides a Grid File Access Library (GFAL), which is a C POSIX-like API that includes functions such as gfal_open, gfal_read, etc. GFAL provides interface to different existing SRM implementations for file access and to gridFTP for file transfer.

LCG File Catalogues (LF): Mapping of LFNs or GUIDs to SURLs is stored in the LCG File Catalogs (LFC). A LFC uses either Oracle or MySQL as a backend database, and is integrated with GFAL mentioned above. The LFC exposes methods to its clients through the GFAL interface that, in turn, interacts with the SRM implementations (for file access) and gridFTP (for file transfer). The LFC client has a POSIX-like command line interface with commands such as lfc-chmod, lfc-ls, lfc-rm, etc.

2.3.2 The AliEnFS File System

AliEnFS [5] is a Linux File System for the AliEn (ALICE Environment [20]) Grid services. AliEnFS integrates the AliEn virtual file catalogue into the Linux kernel via the Linux Userland File System (LUFS) [11] . The AliEn file catalogue allows transparent access to distributed data sets using various file transfer protocols. Thus, AliEnFS makes the file catalogue available for the Grid users and applications under the Linux operating system as a new file system.

Like in the gLite file system framework described in the previous section, files in the AliEnFS are identified by their logical file names (LFNs), which are mapped to corresponding physical file names (including access protocols) in the AliEn virtual file catalogue. Directories and subdirectories are connected like inodes in an ordinary file system. As reported in [5], the AliEn file catalog uses MySQL DBMS as a backend database. Virtual directories are represented as tables in the database where subdirectories are linked to directories via sub-tables entries. While an LFN is mapped to a single “master” PFN, locations of replicas (if any) of the file are stored in a dedicated table that contains mapping of LNFs to replica locations. An additional table keeps file metadata, i.e. mapping of LFNs to file attributes such as size, owner, group, and file access permissions.

As already mentioned, implementation of the AliEn virtual file system under Linux is based on LUFS (Linux Userland File System) that has a modular file system plug-in architecture. LUFS includes a generic LUFS VFS (Virtual File System) kernel module, which can communicate with various user-space file system modules. The kernel module redirects VFS calls via UNIX domain sockets to a LUFS-daemon in the user-space, which, in its turn, loads according to the mount parameters, the requested file system module, e.g. the alienfs module for the AliEn file system. Implementation details of the AliEn file system, and also a more general gridfs module that allows to pug-and-play different grid file systems, can be found in [5].

The AliEn file system offers POSIX-like API in C++ for applications and users. To access the AliEn file system, each individual user has to mount it to its local file system. User authentication (actually authentication of the user-level thread) is performed only once during the mount operation. Authentication of users in the AliEn file system is based on certificates.

In AliEn FS, a file is a write-once entity, i.e. applications are assumed to write to a new file instead of updating an existing file.

An access operation to a file starts with the LNF resolution using a dedicated AliEn service that operates with the AliEn file catalogues. The lookup service determines all locations of the requested file given its LFN and

selects the “best” (e.g. closest) location of a file replica. On the open operation, the alienfs client connects to remote server and possibly downloads the entire (or a part of) file to its cache. On the open operation the user permissions are checked on the client and the server sites. The open operation is followed by either read or write, and then by close. In the case of the write operation, on close the entire file, or a part of it, is transferred to the remote server; and a new LFN-PFN mapping is inserted into the file catalogue. The authorization in AliEn is based on ACLs.

As described in [5], a new architecture, called Crosslink-Cache Architecture, for the file transport layer of the AliEn file system as been developed. With this architecture of the transport layer, all file accesses are routed through distributed cache site-servers to use the server caches in order to improve the performance, availability and scalability of the file system. Authors claim that with this architecture, for each file access, a Grid service will provide information about most “efficient” file access route. In particular, this service can allow automatic file replication by routing write operations through caches. Use of the distributed server caches also enables load balancing. For the time of publication of [5], the work on the -Cache Architecture was in progress.

2.3.3 The PUNCH Virtual File System

The PUNCH virtual file system is a proxy-based virtual file system for computational grids. The PUNCH VFS allows transferring data on demand between storage and compute servers. It is based on the standard NFS server and client protocols, plus software proxies that act as brokers. It has several main features like user transparency and on-demand data transfer. It also does not require modification of the application and it relies on native file systems.

PUNCH also introduces the concept of logical user accounts. The motivation is that data and applications are usually tied to local administrative domains therefore users need real accounts on every single machine which they want to have access to the resources. So, automatic resource management and allocation will be complex because of management of user accounts and access rights.

Another problem is that each local administrative domain might add, remove or modify user accounts and access rights that affect other domains in the Grid, therefore these changes must be propagated in a proper time. Local domains also use policies to control access to their resources. These policies might be dynamic and depend on some real time events. It is difficult to manage these policies with permanent user accounts that have direct access to the resource.

Finally, local file systems like NFS assume that users have unique local Ids, which might cause scalability problem in the Grid.

To address the above problems, PUNCH uses an abstract layer between the physical layer (local resources) and the Grid. This layer includes two major components: Logical user accounts and a virtual file system.

A logical user account consists of two parts: Shadow account and File account. Shadow accounts consist of user ids on compute servers (uids in Unix) that assign to the user dynamically when needed. Shadow accounts will be reclaimed by the system when the user session is completed. File accounts are used to store user’s files. Users access the files through virtual file system. Different user files can be multiplexed into one file account that contains sub directories for each user. One of the main tasks of the virtual file system is to map shadow account to file account for each user.

PUNCH performs authentication using extra proxy called intra-proxy between user level and privileged proxy. These proxies are transparent to NFS and are parts of GRID middleware. The inter proxy controls the transactions between the user proxy and the privileged proxy. This process ensures that privileged proxies response only to requests from authorized user proxies.

2.3.4 Grid Datafarm (Gfarm) File System

The Gfarm file system [21][22][23] developed within the Grid Datafarm project is a world-wide-range virtual file system that provides transparent access to dispersed file data in a Grid accessed by different network file and file transfer protocols. The file system is based on mapping from virtual directory tree to physical files. The Gfarm supports file replication for load balancing and fault tolerance. The Gridfarm file system available at [21] can be referred as one of the reference implementations of the OGF (Open Grid Forum) Grid file system proposed and considered in the OGF Grid File System Working Group [24], [25].

The Gfarm FS provides global virtual name space rooted at /gfarm. The file system is based on two major components: Gfarm filesystem nodes and Gfarm metadata servers. Each Gfarm filesystem node serves as both I/O node and a compute node with a large local disk(s). Each node executes the Gfarm File System Daemon (gfsd) that processes remote file operations with access control as well as performs user authentication, file replication, fast invocation, node resource status monitoring, and control.

Like in the AliEn FS [5], a Gfarm file is a write-once entity, i.e. applications are assumed to write to a new file instead of updating an existing file. Gfarm files are divided into fragments which are distributed across the disks of the Gfarm File System; this allows parallel access to the files. A file fragment has an arbitrary length and can be stored on any node, transferred and replicated in parallel (independently). Gfarm files can be replicated (on the fragment level) for backup and load balancing. The Gfarm FS Metadata servers keep track of file replicas. Gfarm file replicas can be transparently accessed by a Gfarm URL as long as at least one replica is available for each fragment of the target file. If there is no replica, Gfarm files are dynamically recovered by re-computation using access history stored in the Gfarm Metadata Database.

Clients access Gfarm files via the Gfarm parallel I/O API that provides a local file view in which each processor (client) operates on its own file fragment of the Gfarm file. The local file views are also used for newly created Gfarm files.

The Gfar FS metadata are stored in the Gfarm Metadata Database. The database contains different information including mappings from Gfarm logical filenames (LFNs) to physically distributed fragment filenames, a replica catalog, platform information such as the OS and CPU architecture, file status information (e.g. file size, protection, modification time, etc.) and file checksums. It also keeps access history that is used to re-compute the data when a node or a disk fails and to indicate how the data was generated.

2.3.5 Legion and Avaki

Avaki[43] is a commercial Grid solution that claims to address the issues of security, providing a global name space, fault tolerance, accommodating heterogeneity, scalability, persistence, extensibility, site autonomy and complexity management. Avaki's core design principles are providing a single system view, providing transparency as a means of hiding detail, providing flexible semantics, providing reasonable defaults, reducing "activation energy", do not require changing host OSs and network interfaces, and do not require to run Grid in the privileged mode. Avaki includes a so-called "Avaki Data Grid"[45] which is based on the LegionFS file system[44] developed earlier during the Legion project. The LegionFS offers location-independent naming, security with ACLs, scalability, extensibility and adaptability. LegionFS (and Legion itself) is ".. an object-based system comprised of idependent, logically address space-disjoint, active objects that communicate via remote procedure calls (RPCs)"[44]. LegionFS provides abstractions encapsulating files in Unix file system. The Avaki data grid provides a service that allows remote clients to access the data grid using the NFS protocol, and, thus, existing NFS clients.

2.4 File Transfer

File transfer is an important service of data management in Grids that is used by many other Grid services and components such as replica management, a Grid file system (storage), resource allocation and

scheduling, job management. File transfer allows moving files to the location where the files are accessed and processed. GridFTP, shortly described below, defines the state of the art for file transfer in Grids.

2.4.1 GridFTP

Defined by the Open Grid Forum, GridFTP is a data transfer protocol that extends the standard FTP protocol, which is the most commonly used protocol for data transfer on the Internet. GridFTP provides secure and high performance data movement in grid systems.

There are several implementations of the GridFTP protocol but the most commonly used one has been provided by the Globus Toolkit (GT). GT GridFTP defines the state of the art for file transfer in Grids. This protocol has several features like support for Grid Security Infrastructure (GSI), third-party control and data transfer (initiated by site A, a file transfer between sites B and C), parallel data transfer using multiple TCP streams, Striped data transfer using multiple servers, Partial file transfer and support for reliable and re-startable data transfer (automatic restarting after a network failure) and automatic negotiation of TCP buffer/window sizes. It also supports the following URL prefixes (schemas): file://; ftp://; gsiftp://; http://; https://.

A URL, specified as an argument in the file transfer operation, is used to identify the target file (or directory) on the destination server. For example, the URL that points to the file /home/user/grid4all/doc.txt on the "server s1.kth.se" is gsiftp://s1.kth.se/home/user/grid4all/doc.txt

GridFTP operates under a client-server model, where the server runs at a remote site. One improvement to the regular FTP protocol that uses two TCP connections between the client and the server (one for control messages, and another for actual data transfer) is that GridFTP introduces the possibility of multiple data connections that improves efficiency of data transfer. GridFTP is used mainly for transferring data between applications, but its additional commands (for example list files and make directories) allow GridFTP servers to be used as secure data repositories.

Like the regular FTP protocol, GridFTP is a session protocol. The host which provides the service must listen on a known port and wait for clients to send a request. Therefore a daemon must be running and listening for incoming connections. This daemon creates and starts a GridFTP server process when a new connection is received. After connection is established, the connected client can send commands to the GridFTP server. The following components are involved in transfer:

- Security component: This component provides authentication and encryption over control channel.
- Setup component: This component specifies some information about the transfer which can be set/get by the client, such as the type of the file (Binary or ASCII), the MODE of the transfer, the size of a file before transferring it, etc.
- Data channel: This component provides information and negotiation for the data channel. The transfer can be done using store (STOR), retrieve (RETR), extended store (ESTO) or extended retrieve (ERET).

Globus provides two libraries to access the protocol:

- **The Globus FTP control library:** It provides low-level services needed to implement FTP client and servers. The API provided by this library is protocol specific.
- **The Globus FTP client library:** It provides a convenient way of accessing files on remote FTP servers.

GridFTP security is based on X.509 certificates and proxies. GridFTP can work with both full and limited proxies. In order to access a particular file on the server, the client should be mapped to a server side account and this account should have enough privileges to access the file.

There are different modes for authentication in GridFTP:

- **No authentication (N):** No authentication handshake will be done upon data connection establishment.
- **Self authentication (S):** A security-protocol specific authentication will be used on the data channel. The identity of the remote data connection will be the same as the identity of the user which authenticated to the control connection.
- **Subject-name authentication (S):** A security-protocol specific authentication will be used on the data channel. The identity of the remote data connection must match the supplied subject-name string.

2.5 Replica Management

One of the state of the art mechanisms in replica management is replica management in GT4. Replica Location Service (RLS) and GridFTP are the basic data management services involved in replica management in GT4. RLS provides access to information about the location of replicated data. Higher-level services such as Reliable File Transfer (RFT) [9] and Data Replication Service (DRS) [10] use GridFTP and RLS.

Replica management in GT4 is based on the Giggle framework [8] that provides a hierarchical distributed index of replicas in which each replica is identified by a pair of a logical file name (LFN) and a physical file name (PFN). Local Replica Catalogs (LRC) store the PFN-to-LFN mappings. Replica Location Indices (RLIs), which collect the mapping information from LRCs, are organized into a hierarchical distributed index. Information in RLIs is periodically refreshed by LRCs according to a soft-state-update protocol.

The Giggle-based replica location framework in GT4 has a certain limitation: LRCs and RLIs are deployed statically and must be reconfigured manually whenever the system configuration changes. It is also not easy to recover from RLI failures.

2.6 Discussion

Many works have been done in ordinary file systems as well as distributed, Grid and P2P file systems. However, most (if not all) of their design and implementation are based on assumption of stable and robust infrastructures (pool of file servers and storage devices) which is a unrealistic assumption for Grid4All scenarios. The observed systems do not provide sufficient support for self management in dynamic infrastructures. To provide self management, VOFS components should have monitoring and control interfaces that would allow plugging in self management logic (controllers) in VOFS.

3. Requirements and design issues for VOFS

Virtual Organization (VO) is a dynamic federation (run-time entity) of heterogeneous organizational entities (individuals, organizations, institutions, and resources) united by some common interest or task, and sharing data, metadata, and processing and security infrastructure [6].

Virtual Organization File System (VOFS) is a virtual distributed file system that aggregates data resources (file systems and disk space) exposed by VO members, in order to provide seamless access to those objects for VO members and their applications, making an illusion of a distributed file system such as NFS (Network File System). VOFS is a one of the data-oriented services that could be provided in Grids, e.g. replica management, file transfer, etc. VOFS should provide transparent access to data objects (files and directories) exposed by different VO members from different administrative domains. VOFS should hide the heterogeneity of aggregated file systems and security mechanisms used to control access to exposed data objects.

The most important difference of VOFS from ordinary distributed file systems such as NFS and AFS, is its VO-awareness, i.e. it aggregates files from different administrative domains in a Virtual Organization that has own security model and security mechanisms for authentication, authorization and accounting. VO support notion of membership, roles, and policies governing who can (or is obligated to) do what [7]. Data objects (files and directories) in VOFS are accessed by VO members and their applications that act upon VO security policy and mechanisms used in order to control access to data resources exposed by VO members and data services provided by the VO. Thus, one of the important challenges in design of VOFS is to provide consistent and efficient interaction of the VO-level security model and mechanisms with local security mechanisms of administrative domains forming the VO as well as with external administrative domains, providing resources and services for the VO.

3.1 Requirements

Using VOFS, VO members can share their files with each other and provide access to those files for VO applications. A Virtual Organization File System (VOFS) should aggregate heterogeneous native file systems exposed by VO members to VOFS.

There are several important requirements to be fulfilled when designing and developing VOFS, as listed below:

1. *Aggregation of heterogeneous file systems*: VOFS should be able to aggregate multiple native file systems of different types and protocols. VOFS should enable the plug-n-play of heterogeneous FS and data storage resources. A VO member should be able to expose her local file and storage to VOFS and should be able to mount VOFS to her local file system to make it accessible for her operating system and applications. A file can be exposed to several VOFS at the same time.

2. *Operate in the presence of churn and high dynamicity of VO*: In highly dynamic VOs, nodes as well as VO members join and leave frequently. VOFS should have a mechanism to provide a persistent (highly available) and consistent file system that can deal with this high dynamism. The lifetime of VO itself also could be long or short. Some VOs might be created and stay alive for very short time while other VOs might stay alive for long time. VOFS should provide support for both cases.

3. *Operate across different administrative domains*: VOFS usually consists of different administrative domains. Each domain has its own specific systems like security, access control mechanisms, protocol etc. VOFS should be able to aggregate file systems from these domains and provide a general uniform system which can interact with each different domain.

4. *Self-management*. As FS and storage can be dynamically added/removed, VOFS should be self-organizing and self-managing so that adding or removing data resource should be managed without much efforts (if at all) of VO members or human administrators.

5. *Convenient to use APIs (to some degree familiar to users)*. VOFS should provide familiar to users IEEE standard POSIX file API, e.g. open, close, read, write, mkdir, etc. With the ability to mount VOFS to a local file system, the standard POSIX API should allow to support existing clients and applications, e.g. Windows explore and Acrobat Reader, other legacy applications, as well as to develop new applications that uses files. VOFS should also provide an API to data objects with specific semantics, e.g. multilogs.

6. *Operate under VO security (authentication, authorization and accounting)*. VOFS as one of the VO resources should operate under the VO security policies and use Grid security infrastructure to enforce the policies. The Grid security infrastructure should provide interoperability of the VO-level security mechanisms (including authentication, authorization, accounting, and encryption) with various local security mechanisms used in real organization forming a VO.

7. *Support for explicit disconnected operations and reconciliation*: Because of the high dynamism feature of VOs VOFS should provide mechanism to let user to continue to work even after he/she is disconnected from the VOFS. When user is connected again, VOFS should provide a best effort consistency mechanism synchronize the files that modified offline with the online files. The stronger forms of consistency should be delegated to higher-level applications like semantic store.

8. *VO membership*: VOFS should provide a membership mechanism to allow users join and leave the VOFS and make user groups. Groups can be used in assigning access rights to resources. A user might be member of several VOs at the same time.

9. *Scalability*: The size of a VO might be small or large. Number of users in a VO might be tens while in another VO might be thousands. The VOFS should support this variety and have same performance despite of the size of the VO.

3.2 Design Issues

Some of the design issues listed below (e.g. aggregation, self-management, and security) are specific for VO-aware FS, i.e. VOFS; whereas other issues are rather common for ordinary distributed file systems that provide access to remotely located files based on the same communication and access protocol with the same security mechanism.

1. *A global (federated) hierarchical name space*. Each VOFS should have its own name space and should belong to only one VO. However, a VO may have several VOFS. A name of a file in the VOFS name space is logical name, which can be mapped to several physical replicas of the file. Multiple logical names can point to one physical file. Data objects (files and directories) exposed to and created in VOFS need to be uniquely identified within a VO. On the other hand names to exposed and new data objects should be given by VOFS users (humans and applications). In order to achieve this, VOFS should provide a global (federated) name space for VOFS objects with POSIX operations on logical names, familiar to users file/directory attributes, ability to make soft symbolic links like in ordinary hierarchical file system. VOFS should provide at least simple file search operations (e.g. find by name), and it should also provide more sophisticated (e.g. find by attributes, keywords) search operations.

2. *Support for data objects with specific semantics.* Grid4All will provide support for collaborative applications in the form of the semantic store.

3. *Latency tolerance mechanisms* such as caching, prefetching that allow to reduce network traffic; streaming, network coding, and bulk (collative) operations to improve communication efficiency.

4. *Data caching, replication and consistency*. Use of caching (on clients) causes the cache coherency problem, which is typically addressed by using a write-invalidate (callback) mechanism. Most of existing DFS support single-writer open-to-close consistency model and provide lock-based synchronization for mutual exclusion. VOFS should provide also data replication to improve remote file access time (performance) as well as fault tolerance.

5. *Data persistency and availability*. Grid4All assumes high dynamicity of VO members and resources. VOFS should provide a mechanism that should guarantee availability of exposed data objects in presence of churn, i.e. when a node from which data objects were exposed to VOFS goes offline or fails. The data objects can be uploaded to a stable file server (we call it mount server).

6. *Metadata management*. Metadata of a VOFS is a collection of information on objects exposed to VOFS, which includes traditional information on files and directories such as file size, last modification time, creation time, list of files in a directory, etc.; and the VOFS-specific metadata such as local-to-global name mapping, access points (URI) of the exposed objects, access control lists, etc. VOFS should maintain a (distributed) hierarchical metadata catalog.

7. *Role-based policy-based security*. VOFS will use a role-based policy-based security mechanism for authentication, authorization and accounting with language support for policies and authorization.

4. VOFS Architecture

This section presents the design of the Grid4All Virtual Organization File System (VOFS) that is a VO-aware virtual distributed file system that aggregates data resources (file systems and disk space) exposed by VO members, in order to provide seamless access to those objects for VO members and their applications, making an illusion of an ordinary distributed file system, e.g. NFS. VOFS should provide transparent access to data objects (files and directories) exposed by different VO members from different administrative domains. VOFS should hide the heterogeneity of aggregated file systems and security mechanisms used to control access to exposed data objects.

We also define a notion of VO-awareness of VOFS. In particular, it describes authentication and authorization control mechanisms used in VOFS in order to control access to data resources exposed by VO members and data services provided by VOFS, and how those mechanisms interact with local security mechanisms of administrative domains of Grdi4All VO members and external administrative domains, which are not-members of Grid4All, providing data resources for Grid4All.

VOFS security mechanism will be a single sign-on, role-based security infrastructure that uses certificates (credential) for authentication, credential delegation and proxies (to issue and delegate short-term credentials to processes that execute/act on behalf of VO members), and so-called Grid-mapfiles or Community Authorization Services" (CAS) that define mapping of VO members to local users at a resource site. The Policy Enforcement Point (PEP) maintained at a VO-layer at the resource site performs authorization control in Grid4All. The PEP intercepts the access requests from clients and sends requests to a Policy Decision Point (PDP) that checks whether access can be granted or denied given an access operation, a name of a calling client, her role, her access rights to the resource and the VO policy. In the case of VOFS, VO authentication and authorization should be integrated and combined with local access control mechanisms.

4.1 Definition of VOFS

Virtual Organization (VO) is a dynamic federation (run-time entity) of heterogeneous organizational entities sharing data, metadata, and processing and security infrastructure.

VO file system (VOFS) is a virtual distributed file system that aggregates data resources (file systems and disk space) exposed by VO members, in order to provide seamless access to those objects for VO members and their applications, making an illusion of a distributed file system, e.g. NFS. VOFS is a forest of directory trees and files exposed by VO members from file systems of different types and from different administrative domains. VOFS operates under the VO security policy and security mechanisms of participating VO members (real organizations).

VOFS is a part of the Grid4All data storage that can offer the following services for VO members and applications:

- VOFS that is a distributed file system with a standard POSIX API that can be mounted to a local file system as an NFS-type file system;
- (reliable) file transfer;
- data replications;
- replica location service;
- file access monitoring;
- distributed Web server; distributed FTP server; distributed database;
- file sharing;
- backup service;

- semantic store;
- other data services on demand.

VOFS supports several types of data objects, including not only the classical files and directories, but also in the final version of prototype the multilog objects of the Semantic Store layer. The architecture is sufficiently general to support more specialised types such as databases, but this is out of the scope of Grid4All.

VOFS can use different data access and transfer protocols such as HTTP, FTP. We can image a situation when data objects, e.g. files, exposed to VOFS can be accessed only in a standard way with some standard protocol, e.g. FTP, because a Provider (an Owner) of exposed data objects does not want to execute any special (Grid4All) VOFS servers, but the FTP server. The Provider might suspect that the VOFS server contains security "holes "and therefore she does not trust the VOFS server. In this situation, we might need to implement a NFS-like distributed file system on top of FTP, in order to expose those files to VOFS.

In this document, we consider only files and directories (and disk space) as data objects in VOFS accessed with the NFS protocol (NFS-like protocol).

4.2 Building VOFS

A VOFS should provide illusion of a distributed file system that can be mounted to the local file system as an NFS-type file system. We need to provide ability to mount VOFS to a local file system so that VO members and their applications can transparently access files of VOFS as if they were on a local file system.

A VOFS is built by aggregating data objects (existing files, directories and disk space) exposed by VO members and by creating new files and directories on the exposed disk space. Each VO has its own VOFS structure. A data object might be exposed to several VOFS instances at the same time, under the same name or different names. The process of aggregating existing data objects in a VOFS is called *exposing* data objects to VOFS. Objects (files and directories) can be exposed to any directory of VOFS, which we call a *target directory* (or *expose directory*). An exposed data object becomes a part of VOFS. Note the exposed data object is not physically moved or copied from its home location to a new location but rather virtually linked to the VOFS. In general, the Owner can expose not only files stored on her local disks, i.e. local files, but also remote files of a distributed file system that might be mounted to her local file system.

When an Owner exposes a directory, she implicitly exposes disk space by allowing other VO-members to create new files and directories in the exposed directory. A disk space can be explicitly exposed by exposing a directory to VOFS. Disk space can be also consumed by adding more data to an existing file. To give the Owner ability to control the amount of disk space exposed, VOFS should provide a disk space quota mechanism.

A resulting VOFS is a forest of file trees exposed from different sites (computers) within one VO. VOFS has a hierarchical structure of directories and files rooted at a (virtual) root "/". In the section below that describes VOFS bootstrap we consider possible design options of hosting the VOFS root directory "/".

For example, the VOFS depicted in figure 1 is constructed of files and directories exposed from four sites: Site A, Site B, Site C, and Site D, as shown in figure 2. Site A is hosting the root directory of the VOFS and provides access to its local files under directory `/vo1`; whereas sub-directories `/volcano` and `/documents` are remote references (links) to directories `/docs` and `/volcano-sim` exposed from Sites B and D, respectively.

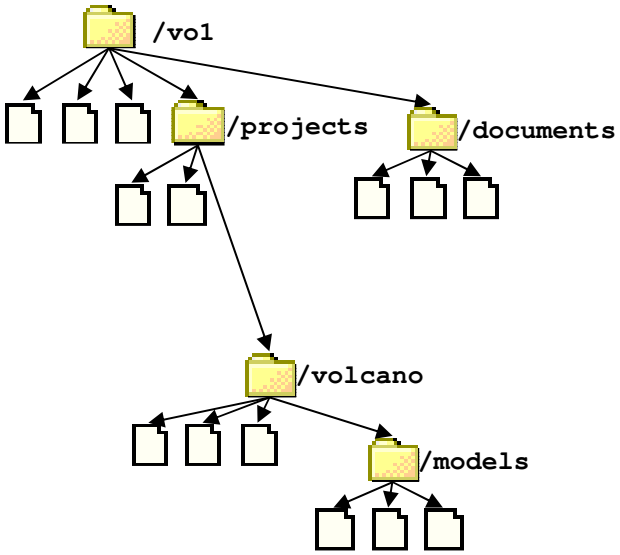


Figure 1: A VOFS

We distinguish between *local* and *remote* objects depending on whether they are exposed from a given site. For example, for the site B, all files exposed from that site are local, whereas the sub- directory `/vo1/projects/volcano/models` (or `./models`) and the parent directory `/vo1/projects/` (or `../`) are remote objects at the site B.

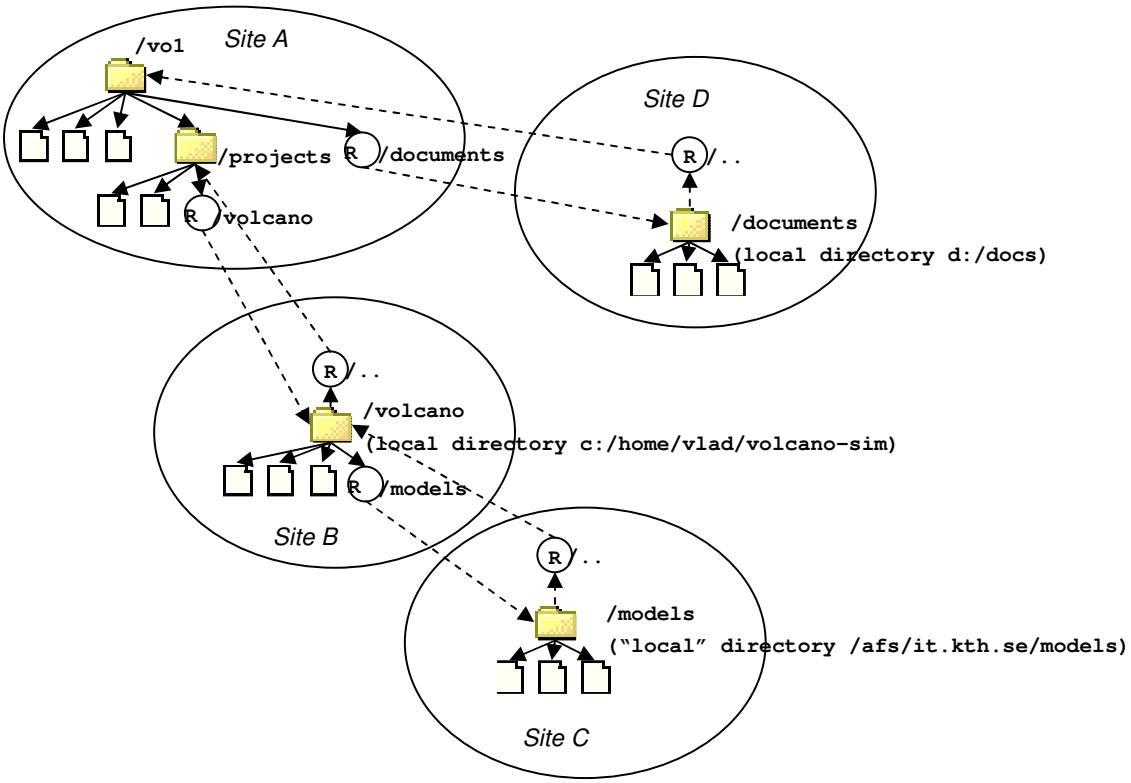


Figure 2: A VOFS structure with remote references (links) to exposed data objects

4.2.1 Exposing Existing Files and Directories to VOFS

If a VO member wants to give an access to her data objects to other VO members and their applications, she exposes the objects to VOFS under a specified VOFS directory (a target directory). It is possible that one object is exposed to several VOFS instances.

When exposing a data object, the Owner specifies which data object she wants to be exposed and where in the VOFS directory tree and under which name she would like that data object to appear. The exposed data object can be given a new name in VOFS. If the target directory already contains an object with the same name, the expose operation fails and reports the name collision error. The expose operation also fails if the target directory is not there, i.e. cannot be accessed.

For example (see figure 2) , the local directory `c:/home/vlad/volcano-sim` on Site B is exposed to VOFS as `/vo1/projects/volcano`; the local directory `d:/docs` is exposed to VOFS as `/vo1/documents`; the directory `/afs/it.kth.se/models` is exposed to VOFS as `/vo1/projects/volcano/models`.

When exposing data objects, the Owner can also specify ACL (an access control list) that defines access rights of VO members on the exposed objects, e.g. directories. An ACL defines who can do what with a data object for which the ACL is set. The VO management and the Owner should agree on ACL or/and Grid map that maps VO members to local user accounts at the object cite and used to control access to the exposed objects.

At the level of VOFS, we can consider definition of access rights similar to those used in the Andrew File System (see Appendix for a short description of ACL mechanism used in AFS). Note that for the sake of efficiency, AFS ACLs are applied and exist at the directory level, e.g. ACL is applied to all files in a given directory.

We assume that to expose files, the Owner will be using a VOFS client that we call *VOFS Explorer* (to be developed) or a special utility client *expose* (to be developed). The VOFS Explorer (similar to Windows Explorer) should allow to browse VOFS, to expose, copy, move, delete, and rename files and directories, to open or edit files with appropriate applications. A special Expose Wizard (a part of VOFS Explorer) will support the expose operation.

We also provide ability to mount VOFS on a local file system so that existing file clients and applications such as Windows explorer can be used to access files in VOFS.

We assume that each site from which data objects are exposed is responsible to start a Grid4All VOFS server process (VOFS demon) that provides access to the data objects exposed from that site. The server is also responsible to maintain a policy enforcement point (PEP) to control access to the exposed files, e.g. for mutual authentication of a client and the server, and for access control.

An object can be exposed from a distributed file system that already includes native servers to provide access to remote files but those servers are VO unaware. We should study possibility of using existing native file servers in VOFS.

It may happen that a directory tree exposed to VOFS from some site contains remote directories attached (mounted) to the tree in a native distributed file system. Access to those remote directories should be provided by the native distributed file system. In this case, the VO-level authentication and access control is performed at a PEP maintained on the VOFS server indicated as an access point for that directory.

For example, a VOFS server on Site B (see figure 2) is responsible to provide access for files exposed from Site B, i.e. it provides access to its local files under the local directory `c:/home/vlad/volcano-sim`, which is exposed to VOFS as `/vo1/projects/volcano`.

4.2.2 VOFS Metadata

Metadata of a VOFS is a collection of information on objects exposed to VOFS. These metadata, in particular, include traditional information on files and directories such as file size, last modification time, creation time, list of files in a directory, etc. The ordinary metadata can be obtained from a VOFS server responsible for a given exposed object. The server can retrieve this information from a local file system using a native file API.

As VOFS is a virtual distributed file system, the VOFS-specific metadata include the following information (but not limited to):

- Expose point that defines where in VOFS directory tree the objects are attached (exposed);
- Access points to the objects, e.g. IP addresses and port numbers of VOFS servers;
- Access control lists; ownership; Gridmaps[1] (if any).

These metadata, in particular, allows finding access points to VOFS files and directories identified by their names in the VOFS tree hierarchy; to browse the VOFS directory tree. Further we discuss how to store, maintain and query VOFS-specific metadata mentioned above.

By analogy to a UNIX file system, we call *i-node* (or *inode*) a data structure holding VOFS-specific information on an exposed data object (file or directory). Each inode contains information including the type of the object (file or directory), access point (or end point reference) to the object, e.g. IP address, port number and an optional access protocol of the VOFS server responsible for the object, e.g. tcp://130.237.212.6:9700. It also has information about the type of the native file system from which the object is exposed, e.g. NFS or DFS, optional access points to native file servers (if any) to access the file using native file system protocol and native clients, information on owners of exposed files and access control lists and Gridmaps.

We assume that metadata (inodes) are collected, stored and maintained by a special VOFS information service that we call Metadata Database (MDDB)[2] [ICCS] that can be either centralized or distributed. MDDB is an index of exposed data objects. MDDB, in particular, provides a lookup service to find a VOFS server responsible for some exposed directory given fully-qualified name of file in that directory. For example given the file name `/vo1/projects/volcano/models/myModel.java`, the lookup query to MDDB should return an access point to VOFS server on Site C, from which the directory sub-tree `/vo1/projects/volcano/models/` is exposed.

There are different ways to design and implement a metadata server. Each one has its advantages and disadvantages. In the following we explain three different approaches for comparison. These approaches are centralized, distributed directory and distributed hash table.

A centralized MDDB stores metadata on all VOFS objects (i.e. inodes) in one place; whereas a decentralized (distributed) MDDB is a distributed index of exposed data objects that can be organized either as a Distributed Directory of inodes or as a Distributed Hash Table of inodes.

In the first version of our prototype we have used the centralized MDDB. But we believe that due to problems of a centralized server (single point of failure, scalability) the distributed directory is better than the others. Using DHT is very similar to centralized approach and the whole tree structure will be stored in a DHT, while in distributed directory each node is responsible only for a part of the tree.

---

[1] Gridmap is a simple list on the resource which keeps information about grid users. The users have unique names and they are associated with the local user names (credentials).

[2] It might be better to call it file catalogue by analogy to other Gird file systems.

Each VOFS is correspondent with only one VO. Therefore it is natural to have several VOFS at the same time and some nodes might be participating in more than one VOFS. Since the tree structure of VOFS is separate for each VO, in all following approaches we should consider to handle several VOFS.

A Centralized MDDB

In a centralized MDDB, inodes of all exposed objects are stored in one place. MDDB can reside on a dedicated server known to clients querying MDDB. With a centralized MDDB, a VOFS server does not need to know about remote objects exposed under its local directories. If an application (a client) tries to access a file of VOFS mounted to its local file system, it issues a lookup request to MDDB to find a server responsible for a file given the VOFS file name. Obvious advantage of a centralized MDDB is a short (1 hop) lookup latency. Obvious disadvantage of the centralized solution is that MDDB is a single point of failure and a potential bottleneck.

Decentralized MDDB as a Distributed Directory (Tree)

In a decentralized MDDB structured as a distributed directory (tree), idones are distributed among MDDB components in the following way. Each site of exposed objects provides an MDDB component that contains inodes of remote objects known to the site, i.e. remote objects exposed to local directories and remote parent directories, to which local objects are exposed. Distributed MDDB components form a P2P system with a tree structure that corresponds to the structure of VOFS tree. Each MDDB component can directly communicate with a set of MDDB components on sites hosting remote objects. To find a site (a VOFS server) responsible for a file given its fully-qualified VOFS name, the VOFS client initiates a resource lookup procedure similar to the one in the conventional NFS. The lookup operation can be initiated at any node in the MDDB P2P directory. The distributed lookup algorithm is based on traversing the distributed directory either up or down (depending on the initial point) until inode of the requested file is found and information on access point(s) is obtained. Each MDDB node can maintain a cache of lookup results similar to the one in an NFS client. The content of the cache is considered as a hint because there is no guarantee that the content is up-to-date.

Decentralized MDDB as a Distributed Hash Table

The third design option in developing of the MDDB index service is to store inodes in a distributed hash table (DHT) using names under which objects are exposed to VOFS as keys. Like in VOFS with the centralized MDDB, in VOFS with a DHT of inodes, a VOFS server does not need to know about remote objects exposed under its local directories. If an application (a client) tries to access a file of VOFS mounted to its local file system, it issues a lookup request to DHT to find a server responsible for a file given the VOFS file name. Worst-case lookup latency in DHT is log (N); average lookup latency is ½ log (N).

4.2.3 Bootstrap VOFS

Bootstrapping a VOFS is similar to bootstrapping a P2P system when a first peer starts and creates the system so that other peers can join it. The VOFS P2P network is formed of Grid4All VOFS servers running on participating sites (nodes). As mentioned above, VOFS is a collection of aggregated file systems (files and directories) rooted at a root “/”.

In our view, there are two options to bootstrap VOFS, i.e. to create a (virtual) root directory.

In first approach, there is a dedicated bootstrap server which is responsible for the root directory, i.e. it provides a disk space to create new files and directories in the root directory and keeps information (metadata) on objects that are exposed under the root directory; the server is contacted first to obtain information on the root directory when an Owner wants to expose a data object to VOFS first time.

Another approach is that a site (an owner) that first exposes a data object to the virtual root directory "/" becomes a responsible owner of the root directory, i.e. it provides a disk space for creating new files and directories in the root directory and maintains metadata on data objects exposed under the root. There is a bootstrap server(s) that maintains information on the site that hosts the root directory "/". The difference from the first option is that any of sites can host the root.

In order to provide a DHT for storing inodes of exposed objects as described above, the distributed system of VOFS nodes should be organized as a structured P2P system with DHT functionality. There are many proposals in literature on how to bootstrap a structured P2P system and how nodes can join and leave the system.

4.3 Mounting VOFS to a Local File System

In order to provide a transparent and seamless access to VOFS for applications, e.g. Acrobat Reader, and existing files clients, e.g. MS Windows, the entire tree or a sub-tree of VOFS need to be mounted to a local file so that the applications can access the VOFS files using an ordinary POSIX file API of the local Operating System.

As stated in the UNIX manual page on the mount (8) command, "mount attaches a named file system to the file system hierarchy at the pathname location directory…" The major parameters to be specified when mounting a foreign (distributed) file system to a local file system include a type of the file system, the URI of the foreign remote directory to mount and the mount point on the local file system. For example, the following command mounts the remote NFS file system at 130.237.151.1 to the local file system on the `/mnt/xyz` mount point:

```
mount -t nfs 130.237.151.1:/root /mnt/xyz
```

The major difficulties (obstacles) in supporting mounting of VOFS to local file system include:

- Heterogeneity of aggregated file systems;
- Spanning of multiple administrative domains;
- Heterogeneity of access control mechanisms in those domains;
- Necessity of enforcing the role-based VO security policy.

In the first approximation, to support mounting of VOFS to a local file system, we can use existing ordinary file system clients and servers that communicate with each other using a file system specific protocol. Assume for example, that some directory from an ordinary distributed file system, e.g. NFS, is exposed to VOFS from a site S1. A User could mount a sub-tree the VOFS rooted at that directory as an NFS-type file system exported from Site A. In this case, the Operating System on the User computer can use an existing ordinary NFS client to access the ordinary NFS file server on S1, as shown in figure 10.

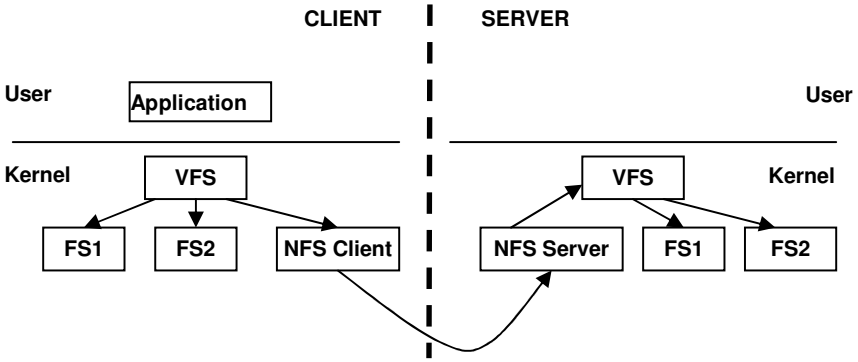


Figure 3: VOFS using existing NFS client and server

However, we cannot assume that VOFS data objects are exposed from local file systems of the same type. Heterogeneity of exposed objects requires providing a plug-and-play mechanism in VOFS that should allow to access different types of file system in a unified way.

As VOFS spans multiple administrative domains, it should provide a security mechanism to control file access across administrative domains in a unified way using role-based security policy. A resource provider (VOFS file server) controls file access rights of requesting clients according to its local policy.

To illustrate the security issue, assume, for example, that VOFS uses an RPC-based protocol to access remote files, similar to NFS. Assume a VOFS file client process of the User U_A in the administrative domain D_A (known as a VO member *A*) makes an RPC to a VOFS file server process in the administrative domain D_B (known as a VO resource *B*). First, the member *A* (as a service consumer) and the resource *B* (as a service provider) have to mutually authenticate each other by exchanging certificates. Next, the resource *B* checks whether *A* is authorized by local policy to access the requested file in D_B, i.e. whether she is allowed to spawn a process to service the call. This two-step procedure (authentication and authorization check) should be performed on the VO-level at PEP (Policy Enforcement Point) on the VOFS server.

The above discussion calls for development of special VOFS client and VOFS server that should provide unified connectivity to file systems of different types (i.e. ability to mount VOFS to a local file system) as well as a VO-aware security mechanism to control access of VO members (VOFS Users) to data objects exposed from different administrative domains.

One way to provide unified connectivity to foreign file systems of different types is to use a loopback adapter as a “remote” file server and to mount VOFS as an ordinary conventional file system, e.g. NFS. We argue for NFS because, to our best knowledge, almost all well-known and widely-used operating systems support mounting of NFS file system (i.e. there exist NFS clients for almost all operating systems).

To use a loopback VOFS adapter, a User indicates *localhost* (instead of an IP address of a file server responsible for the root directory of the mounted sub-tree) in the URI of the mounted VOFS directory, and NFS as a type of the mounted file system. In this case the local operating system will use an existing NFS client to access the loopback VOFS adapter with the RPC-based NFS protocol. The adapter calls the VOFS server that checks access rights of the requesting VO member, and, if access is allowed, the server accesses the native file system on client’s behalf. The VOFS loopback adapter communicates with the VOFS server with VOFS-specific protocol (to be developed). The server converts NFS calls issued by the NFS client in to local calls on the exposed file system as illustrated in figure 4.

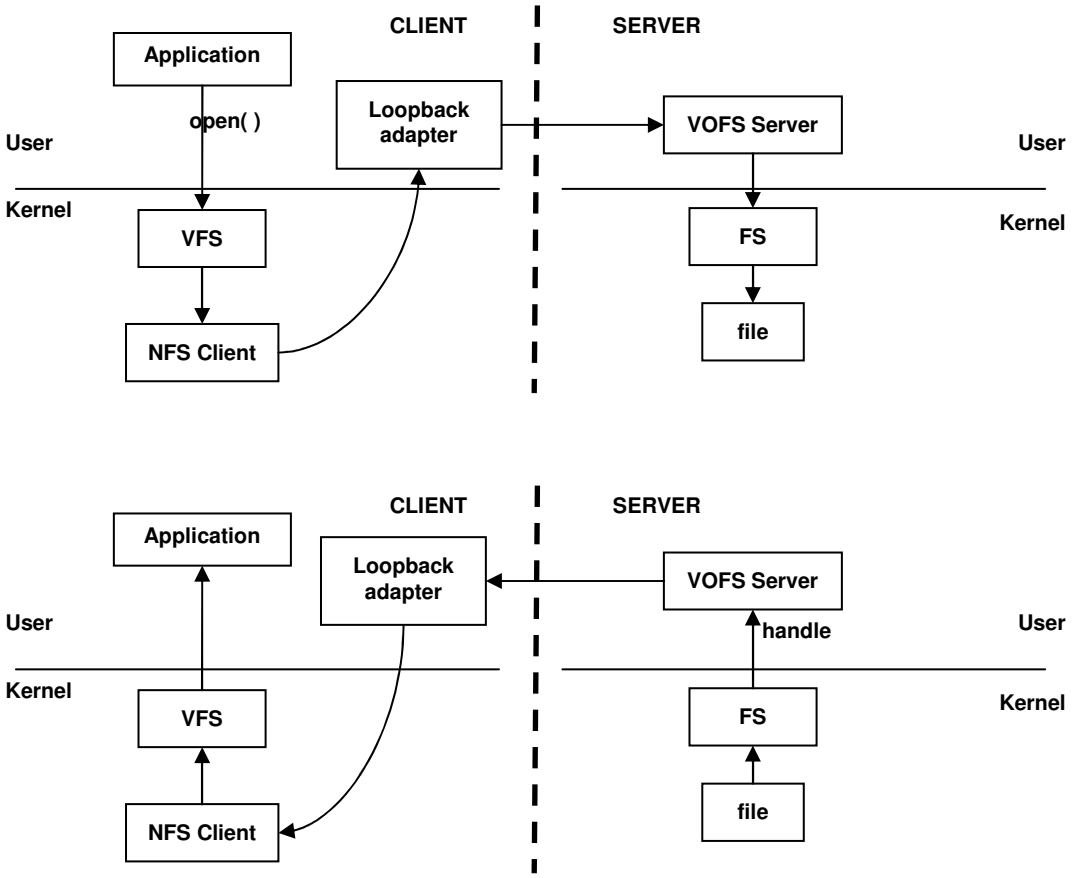


Figure 4: VOFS using VOFS loopback adapters and VOFS servers

In this document, we do not consider design and implementation details of the VOFS adapter and the VOFS server, leaving this to next milestone in M24.

The Distribute File System (DFS) designed in Task 3.1 has a design similar to the one described above. The DFS design is also based on the use of user-space file system, and includes features that fulfills most of the functional requirements to VOFS presented above (e.g. support for disconnect operations). However, the DFS is designed with an assumption that it is deployed and used in a single administrative domain, i.e. it is not VO-aware. The VOFS design presented here addresses the issue to integrated files from different administrative domains, different file systems, as well as to achieve high availability. We expect the two designs to merge.

4.4 Replication and multi logs

4.4.1 Replication in VOFS

In our design we consider two options for replication, transparent replication that provides high availability and explicit replication. There are several requirements in VOFS that can be addressed by using replication mechanisms. In this part we explain how we use replication to fulfil those requirements.

One of the main reasons for using replication is to increase availability. In a highly dynamic environment (like the one we are considering in Grid4All) nodes can join and leave frequently. On the other hand they expose

parts of their local file system to the VO. Therefore if some node joins the VO, exposes some files and then leave, those files will not be accessible to the other members of the VO anymore.

Another problem that might accurse in VO is failures of nodes. If a node joins the VO, exposes some files and then crashes the data will not be available anymore.

To address these problems, we can use replication to create replicas of the actual data on the other nodes or on some servers in the VO so if the original node leaves VO the data will be still available. The underlying mechanism of replication should be transparent to the users but they should be able to specify if they wish their data to be replicated or not. They might do it through some options when they expose their files.

Another reason for using replication is to increase performance. We are also considering using local caching on nodes. When a node gets access to some remote resources (files), it can create a local cache of the file and work with this copy. Then the changes which have been made to the local cache will be propagating to other copies.

Use of caching (on clients) causes the cache coherency problem, which is typically addressed by using a write-invalidate (callback) mechanism. Most of existing DFS support single-writer open-to-close consistency model and provide lock-based synchronization for mutual exclusion.

Caching mechanism is widely used in distributed file systems. There are two modes of caching strategies that are mainly used: write-back mode and write-through mode. In write-through mode, every write operation at client side is immediately forwarded to server. In the write-back mode, client operates a series of write operations on cached data (at client), before flushing it to server. Caching at client side itself varies in different implementations of distributed file system: some cache file data blocks, or data at arbitrary granularity, others like AFS and Coda cache the whole file.

In the current design of VOFS prototype, we adopt whole-file caching at client side. The wanted file at remote client will be downloaded to the local working directory. Client users need to manually trigger the flushing operation of the cached file to replace the original one (master copy) at another client.

As for synchronization, if a client locally modifies a cached file and shortly thereafter another client reads the file from the server, the second client will get an obsolete file. So the synchronization strategy of distributed file system could be essential yet complex. In order to achieve read consistency in VOFS, we employ a synchronization model similar to that of NFS: A master file at a client can only be written by one remote client at a given time and is locked from other reading or writing operations during interval. This intends to guarantee readers to cache a consistent copy of the master file. When it comes to writer competition of multiple clients, so far the prototype simply employs last-writer-wins policy, which means the master file always represents the last modification applied.

In some cases a master file is cached by several clients and they want to keep their local cached copies synchronized with the master file. After finish caching, those clients sends request to master file holder to create call-backs. Whenever a write operation has been performed to this master file, all registered call-backs will be triggered except the writer's. Such call-back includes the endpoint of requestor and both the absolute paths of cached copy and master file. Thus master file holder knows exactly which client to notify and which cached file at that client to replace. So if the call-backs are able to be received by those requestors, they start to receive the latest copy of the master file and replace their local cached copies that are stale. As to the stale cached copy, it can be versioned to another new file with timestamp appended after the file name; also we can set a max number for versioned copies of a single cached file. The call-back registrations can be simply stored in plain text files in hashed form in case of loss. On the other side, Client GUI can serve as observer for call-back notifications and thus pop warning message to notify end user that the cached file has been synchronized.

4.4.2 Support for Multi logs

Semantic store layer needs special objects called multi logs. They keep changes made by each user to an original document. Each user writes locally to his own log in the multilog; his writes are propagated to other replicas of the same log. We believe that extending the basic caching mechanism with explicit replication or cache-aware file API can provide special support for multi logs.

In Grid4All, we consider multi logs support as an application on top of VOFS. The multi logs support should provide its own API on top of an ordinary file system API.

For example, multi logs support ensures that each user gets a local copy of a shared file and has an ability to initiate a reconcile operation. When the user disconnects she works with her local copy of the log. When connected, she can perform the reconcile operation to synchronize her copy with other logs. The multi log API should provide ability to specify semantics in the form of action constraints to be enforced on reconciliation. In order to achieve this functionality VOFS should provide support for specific multilog object.

4.5 VOFS Interfaces (API)

This part specifies the API of the VOFS, which presents the upper level and lower level interfaces.

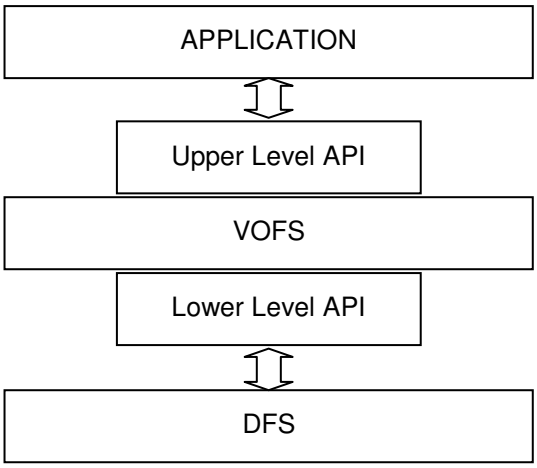


Figure 5: VOFS Interfaces

4.5.1 VOFS Upper Interface

The upper interface specifies the API that can be used by clients of VOFS and other layers and components built on top of it. The API should be compatible with POSIX standard however additional functionality is required to support VOFS specific features.

POSIX API

This API is a subset of standard POSIX API and NFS V4. The following is a list of functions and their input and output parameters.

#	Name	Description
1	ACCESS	It checks what access rights the user has to the resource. Input: The user identifier (User credential) The resource identifier (URI to the resource)

		Output: A bit flag indicating different access rights including: Read, Lookup, Modify, Extend, Delete, Execute.
2	CLOSE	Close the file. Input: Filehandle of the file to be closed. Output: The result of the close operation (if it was successful or not).
3	COMMIT	It flushes cached file on local file system to the VOFS, if the file is cached. Input: Filehandle of the file to be committed. Output: The result of the operation (if it was successful or not).
4	CREATE	Create a new directory (or other objects but not regular files). Input: URI of the place to create the new object Name of the new object. Type of the new object. Output: The result of the operation (if it was successful or not).
5	GETATTR	It gets attributes of an object in VOFS. Input: Filehandle of the object. Output: The set of attributes if it was successful or an error massage.
6	LOCK	It requests for a lock on a specific range of bytes in a file. Input: Filehandle of the file. Offset of the range of the bytes in the file to be locked. Length of the range. Output: The result of the operation.
7	LOCKT	Test if there is a lock on a byte range in a file or not. Input: Filehandle of the file. Offset of the range of the bytes in the file to be checked. Length of the range. Output: If there is a lock it returns the information about it. Otherwise returns OK.
8	LOCKU	Unlock the range of bytes in a file. Input: Filehandle of the file. Offset of the range of the bytes in the file to be unlocked. Length of the range. Output: The result of the operation.

9	LOOKUP	It searches for an object (file or directory). Input: Name of the object Output: The result of the operation. If it was successful it returns the filehandle of the object.
10	OPEN	It opens a regular file. If the file does not exist it will be created. Input: URI of the file to be opened/created Output: If it was successful it returns the filehandle of the file. Otherwise it returns an error message.
11	READ	It reads data from a regular file. Input: The filehandle to the file Number of bytes to be read The offset in the file to start reading Output: If it was successful it returns data in form of a byte array. Otherwise it returns an error message.
12	REMOVE	Delete a regular file. Input: URI of the file to be deleted Output: The result of the operation.
13	RENAME	Rename a regular file. Input: URI of the file to be renamed The new name of the file Output: The result of the operation.
14	WRITE	Write data to a regular file. Input: Filehandle of the file The offset in the file to start writing data Number of bytes to be written A byte array including data to be written Output: The result of the operation.

Table 1: VOFS Upper Interface (POSIX API)

VOFS Specific API

#	Name	Description
1	EXPOSE	User can expose a local object (i.e. file) to the VOFS. Input: The user identifier (User credential)

		URI to the local object to be exposed URI to the target place in VOFS ACL Options (i.e. replication, caching) Output: The result of the operation. It indicates if the operation was successful or if there was an error (i.e. name collision)
2	MOUNT	User can mount a remote object in VOFS as a local object. Input: The user identifier (User credential) URI to the remote object in VOFS Target place in local file system to mount the object Output: The result of the operation. It indicates of the operation was successful or if there was an error.

Table 2: VOFS Upper Interface (VOFS Specific API)

4.5.2 VOFS Lower Interface

This interface specifies the API to the services that VOFS uses. In our design, VOFS needs only standard POSIX API to be provided by the DFS layer.

5. VO-Awareness of VOFS

5.1 Assumptions on the Grid4All Security Infrastructure

VOFS spans multiple administrative domains. We assume that each administrative domain has its own administration and a security policy. When joining a VO, each organization typically retain ultimate control over its policy. The VO may apply its own policy that should not (could not) override or replace local policy decisions. For example, some VO members may have full access to some of VOFS files, while others may have read-only access.

The local security policy can be implemented by different methods, e.g. Kerberos and SSH. Furthermore, the local security policy may be changed dynamically.

We assume that each User (process acting on behalf of the User) and each Resource[3] (process acting on behalf of the Resource) has two names, a global name (i.e. the name in the VO) and a possibly different local name. There exists a partial mapping (Grid map) of global names of Users to local names for each resource in each administrative domain. The mapping is resource-specific and global names can be mapped to local names in various ways:

- Static one-to-one mapping of a global name to a predefined local name;
- Dynamic one-to-one mapping of a global name to a dynamically allocated local name;
- Static many-to-one mapping of a set of global names to a single local “group” name;
- Dynamic many-to-one mapping.

If to consider role-based security mechanism, then a particular identity (a VO member) can be assigned a role via its role attribute. In this case, authorization decision can be made depending of the role of the identity, and it might require an extra check of whether the identity is bound to the role. In any case, use of global names makes it possible to provide single sign-on.

We require from WP2 the following mechanism as a base-line for authentication and authorization in the Gird4All VOFS. The mechanism is a variant version of CAS (Community Authorization Service) in Globus Toolkit 4. It is combined with the GT Grid Security Infrastructure (GSI) based on X.509 proxy certificates to provide credentials for users, delegation and single sing-on.

The VOFS policy is a consistent combination of VO policies, maintained by the VO and communicated to participating sites, with local policies on those sites. A VO policy defines what the individual VO member or a group of members are allowed to do; whereas the local site policy defines what the VO is allowed to do.

In the next section we introduce some of the state of the art mechanisms which can be used in Grid4All to provide security features needed in VOFS.

5.1.1 SAML

The Security Assertion Markup Language [48] (SAML) is an XML based framework that uses assertions to exchange security information (e.g. authentication and authorization information) between different administrative domains. The major purposes of using SAML are the followings:

[3] Resource Provider

Single Sign On (SSO):

If user wants to have access to resources from different security domains she should be authenticated and authorized by each domain, which means user should login to each domain separately. But with SSO, user can login to one domain and the other domains authenticate user automatically. Web browsers use cookies to do SSO. But they work only in one domain and not across multiple domains. SAML solves this problem by providing standards for exchanging security information between different domains.

Federated Identity:

Another problem in multiple security domains is handling different user identifications. One user need to have an identity and associated attributes in each domain. With help of federated identity, domains can communicate with each other and use and share the identification information about users.

There are three major participants in a SAML transaction:

1) *Assertion party*, which is also called Identity Provider (IdP) or SAML authority. IdP identifies the user locally and provides an assertion and sends it to relying party.

2) *Relying party*: which is also called service provider (SP). It receives the assertion and identifies the user based on it and makes access control decisions.

3) *User*: which is also called subject or principal. This is the entity to be identified.

Use Cases:

Use case 1: Single Sign On (SSO)

In this case, user is authenticated by a security domain (e.g. by password) which is called IdP and can have access to resources in other security domains which are called SP. There should be a federation agreement between these domains to share the information about the user identifications. This scenario is called IdP-initiated SSO (figure 1). Another scenario is that user tries to have access to resources from SP that needs authentication and access control. In this case SP will send the user to IdP with an authentication request. After identifying user at IdP, IdP will send an assertion to SP. This scenario is called SP-initiated SSO.

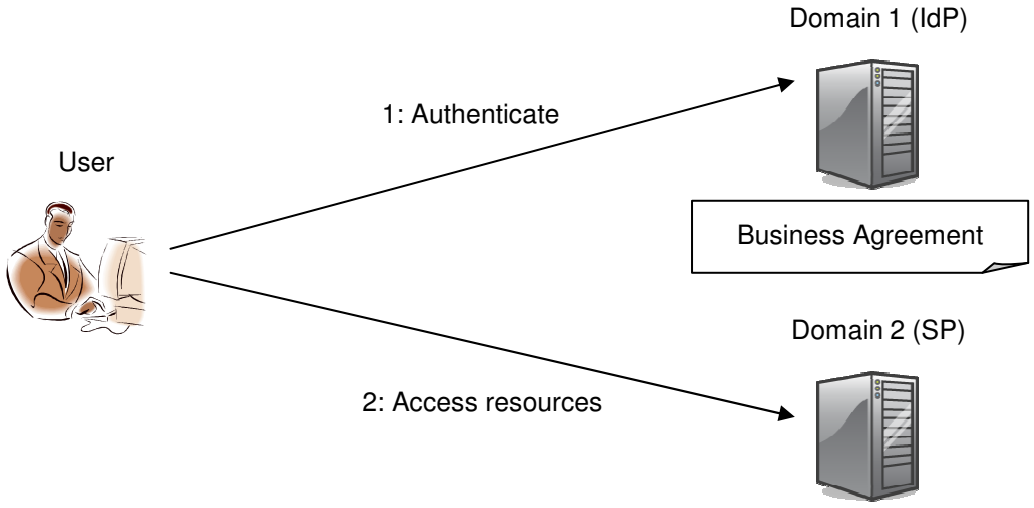


Figure 6: Single Sign On use case

Use Case 2: Identity Federation

SAML provides ability to perform dynamic identity federation between different domains. Assuming one user has different local identities at different domains, in SSO scenario (figure 2) these domains need to have a federation agreement to share those identities.

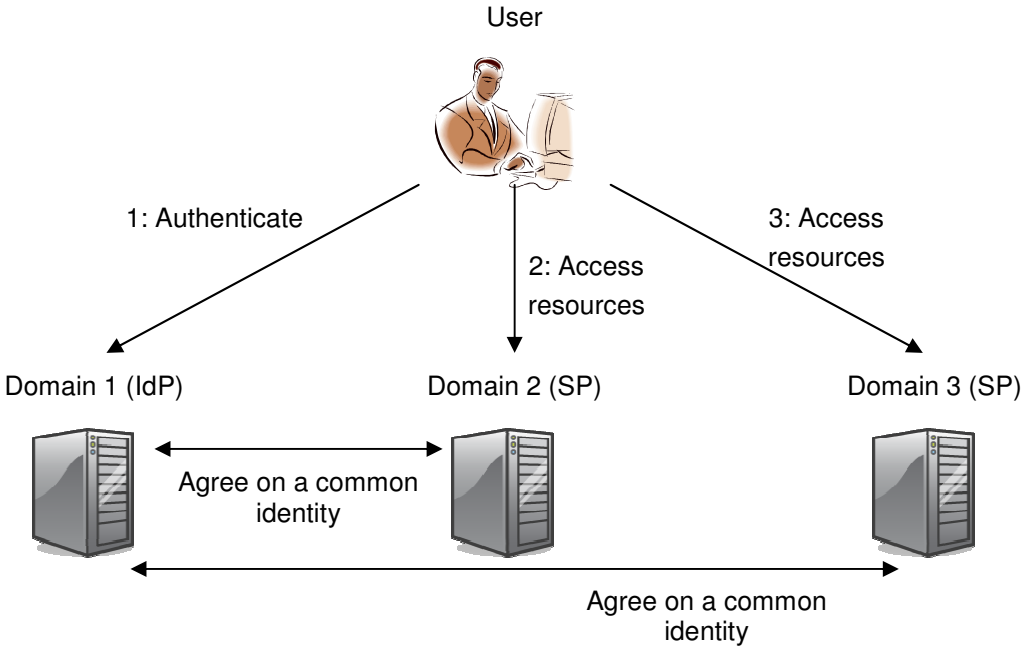


Figure 7: Identity federation use case

SAML Architecture

Figure 3 shows the major components in SAML. One of the basic components is Assertion. Assertions carry statements about a subject (principal) and they are produced by IdPs. Three types of statements in an assertion are authentication statements, attribute statements (name-value) and authorization decision statements (A subject is permitted to perform action A on resource R given evidence E).

Another basic component is Protocol which defines the request and response messages which carry assertions. Next component is binding component. They define the relation between protocols and the underling communication systems like HTTP or SOAP. Some examples of bindings are SAML SOAP Binding (based on SOAP 1.1), Reverse SOAP (PAOS) Binding, HTTP Redirect (GET) Binding, HTTP POST Binding, HTTP Artifact Binding and SAML URI Binding. Finally, the last component is Profile. They define particular use cases by encapsulating assertions, protocols and bindings.

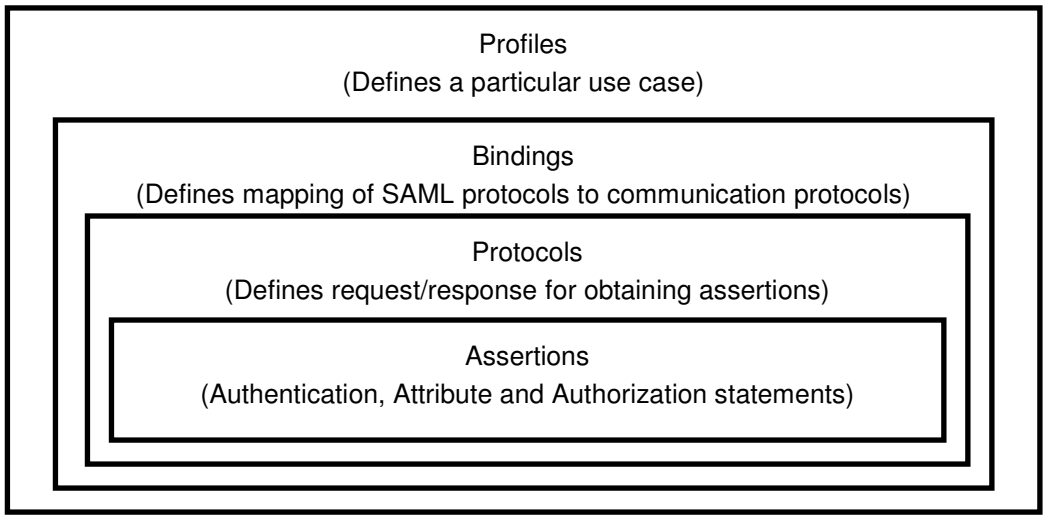


Figure 8: SAML basic components

Using SAML with XACML

The eXtensible Access Control Markup Language (XACML) defines how to express and evaluate policies for access control. Since SAML provides a way to exchange security information, it can be used with XACML. The next figure illustrates how these two systems can be combined:

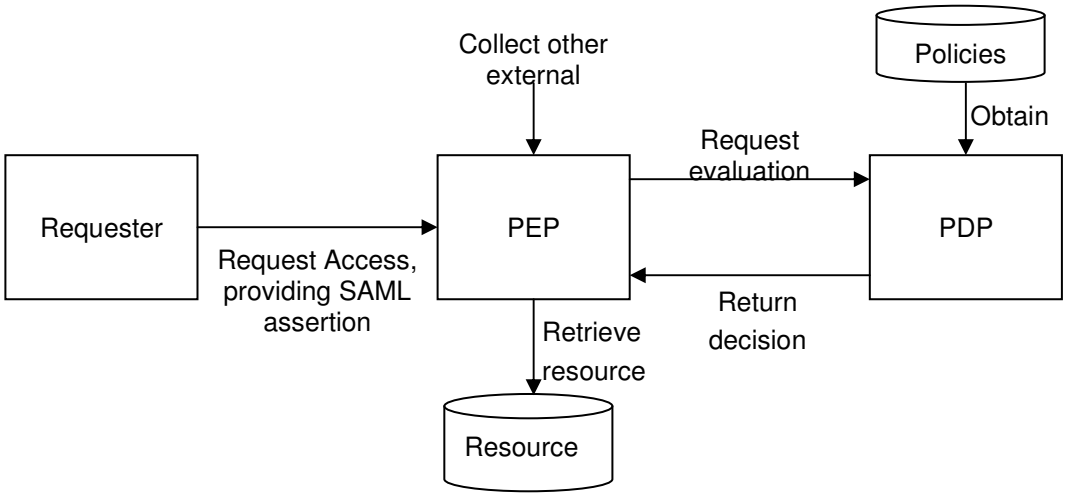


Figure 9: SAML and XACML integration

5.1.2 VOMS

Virtual Organization Membership Service [47] (VOMS) is a framework to provide management of policy based authorization information in virtual organizations. The two major concepts in VOMS are VO (Virtual Organization), which is an entity to group users and resources, and RP (Resource Provider), which provide resources. User information is managed in a server (centralized), but the information about the relation between the user and resources is managed at each RP (decentralized).

Groups and sub groups are presented in a DAG (Direct Acyclic Graph) in VO with the root VO. Users must provide both credentials (for authentication) and authorization information to RP when they want to have access to a resource (Push model). VOMS consists of the following services:

- **User Server:** Provides user information.
- **User Client:** Fetch user information from user server.
- **Administration Server:** Keeps the administrative data (users, groups, roles etc.) in a database.
- **Administration Client:** To be used by administrators to manage information on administration server.

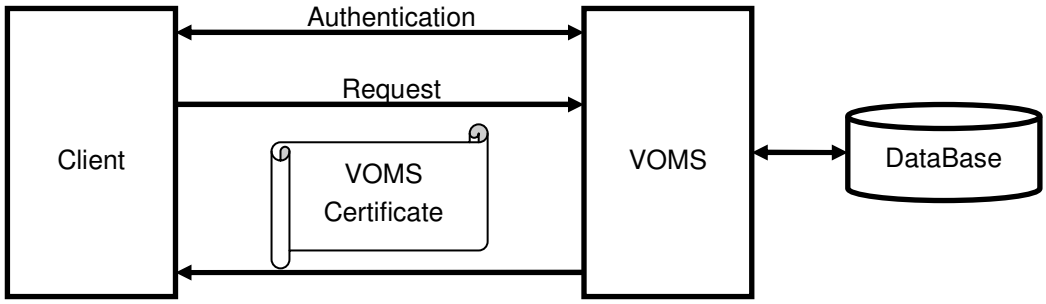


Figure 10: VOMS

5.1.3 Permis

PERMIS [46] (Privilege and Role Management Infrastructure Standards) is a privilege management infrastructure based on hierarchical RBAC model and X.509 for authorization.

Authorization Models:

The traditional model for authorization is DAC (Discretionary Access Control) which users have been assigned access rights to the resources. These access rights are managed in ACL (Access Control List). Another model is MAC (Mandatory Access Control) which every resource has a security label called classification which indicates the security level of that resource, and each user has a list of classifications which is called clearance. User can access resources that she has corresponding classification. Another authorization model is RBAC (Role Based Access Controls). In this model user has one or more roles (e.g. manager, employee etc.). Each role has a set of permissions. When user requests to have access to a resource, she presents her role to the resource. Resource checks the permissions of that role and either accept or reject user's request. RBAC roles can be simple which means there is only one level of roles, or can be hierarchical which means there are nested roles. A typical example of nested roles is like this:

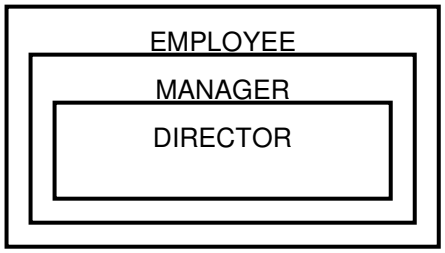


Figure 11: Hierarchical RBAC

It is clear that if permission is given to a role, all its nested roles also will have that permission implicitly. In most systems normally the number of roles is much fewer than number of users. Therefore, RBAC can easily handle systems with large number of users. But DAC systems have scalability problem since for each user an ACL should be defined.

PERMIS has three main components:

- **Authorization policy:** It specifies the access rights of users to resources using policy based authorization.
- **Privilege allocator:** It assigns roles to users by defining attribute certificates (AC). An attribute certificate is a binding between user name and her privilege attributes. ACs are kept in a public LDAP (Lightweight Directory Access Protocol).
- **PMI (Privilege Management Infrastructure) API:** This is the application gateway which accepts requests from users to get access to resources. When it receives a request, the AEF (Access control Enforcement Function) component authenticates the user and then checks with ADF (Access control Decision Function) component if the user can get access to the resource or not. The ADF retrieves policies from LDAP for the roles in the user's AC and makes a decision and sends it back to AEF.

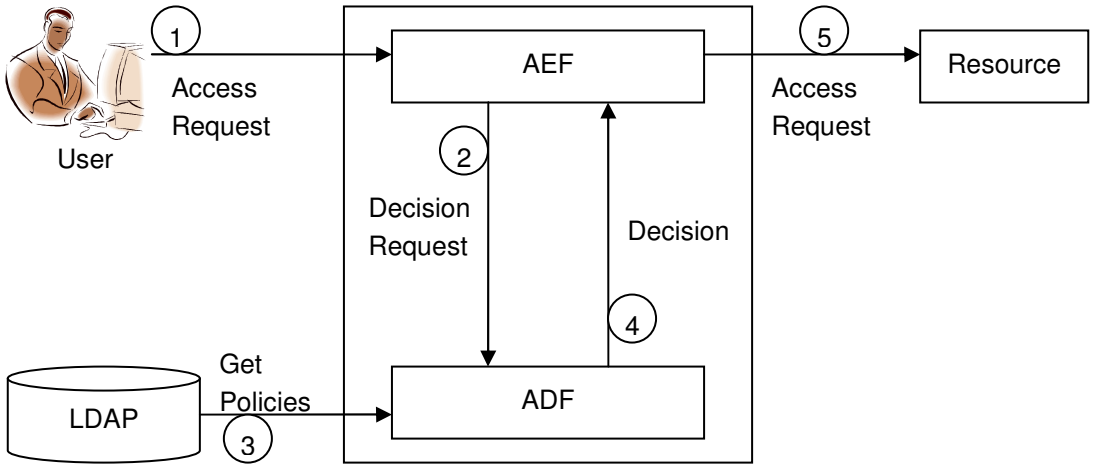


Figure 12: Permis architecture

5.2 Requirements for the Grid4All Security Infrastructure

The VO management system should maintain a list of VO members and the VO's access control policies for its resources including the VO itself (e.g. who can define rights, form groups, etc.). In the case of VOFS, the VO policy, for example, defines which member can read which file.

A Resource Provider (resource site) maintains her policy regarding the VO and VO members using her native mechanism used for local (non-VO) users. The Provider can, for example, map the entire VO to one local identity that represents the VO. In this way, each VO-member (i.e. client presenting a VO certificate) will be mapped to that identity. A VO-member can be also mapped to her individual local identity to enforce site policies regarding that member.

When an authenticated VO-member makes an access request to the resource, the Provider (resource server) first enforces the site's policies regarding the VO as the whole, next, enforces the VO's policy regarding that member, and, finally, enforces that the site's policies (if any) regarding that member. Note that, in this way, site's policies have priority over the VO's policies. Thus, the effective rights granted to the requesting member is an intersection of the set of the rights granted to the VO by the resource provider, the set of the rights granted to the VO member by the VO and (optionally) the set of the rights granted to the VO member by the resource Provider.

When a user wants to access a file hosted by VOFS server, a VO member, as a VOFS client, is authenticated using her certificate that also includes a policy statement of that user's rights. Note that, in general case, the authentication should be mutual. After authentication, the VO member is mapped to a local identity (e.g. a Unix account) of the VO by a local Grid-mapfile. This mapping is the very first access control check: if the VO is not listed in the Grid-mapfile, access to the file is denied.

Once the VO is successfully mapped to a local identity, the VOFS server checks whether the VO (as the whole) is allowed to make the requested action. Then, as described above, the site (the Policy Decision Point, PDP) computes the effective rights based on the policies of VO in respect to the VO member and the policies of the site in respect to the VO member and makes a policy enforcement decision whether that member is authorize to perform the requested action.

6. VOFS Usage Scenarios

This section presents a usage scenario for a VO file system (VOFS). The main aim of designing this scenario is to help developing semantics of the expose and mount operations as well as developing of the MDDB (Metadata Database) component which is a catalog of files exposed to the VOFS. The scenario has been developed with collaboration with researchers from ICCS Georgios Tsoukalas, Aris Sotiropoulos.

6.1 Basic Assumptions

In this scenario we assume two real organizations, KTH (Royal Institute of Technology), Stockholm, Sweden and ICCS (Institute of Communications and Computer Systems), Athens, Greece. Assume two users KTH_User at KTH and ICCS_User at ICCS. Both users use MS Windows as an operating system on their computers. KTH_User stores her files in her local file system on her machine. Assume, KTH and ICCS have created a VO called KTH_ICCS as illustrated in figure 13. Both KTH_User and ICCS_User are members of that VO. Suppose, KTH_User wants to expose some of her files to the VO because she and ICCS_User work on the same project and need to share the files.

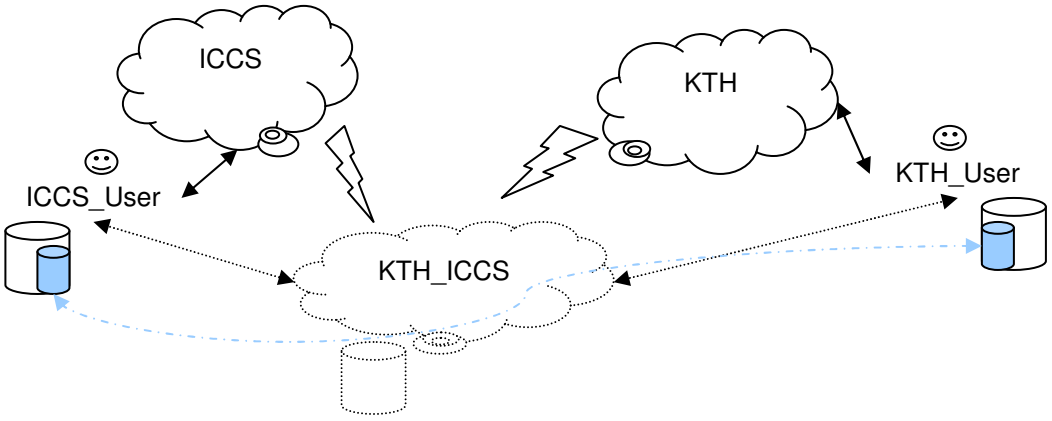


Figure 13: The KTH_ICCS Virtual Organization

KTH-User has a folder `C:\projects\documents\` that she wants to expose to the VO. There are 3 sub folders under that directory:

```
C:\projects\documents\meetings
C:\projects\documents\pictures
C:\projects\documents\maps
```

We assume there is only one file in each of the folders:

```
C:\projects\documents\meetings\meeting1.txt
C:\projects\documents\pictures\pic1.jpg
C:\projects\documents\maps\map1.pdf
```

6.2 Requirements

In this scenario we require that Grid4All VOFS should allow some actions to be performed. The first action is to create one VO with the name KTH_ICCS. Then KTH_User exposes some files from her local file system to this VO. Finally ICCS_User mounts those files into her local file system.

- Both users should be able to see the same directory structure and files, and they should also see the changes made by each other. However, for now, we assume that the Users should not be able to modify/write the same object (file) at the same time, or, there must be a mechanism to find write collisions and inform users about it.

6.3 Description

KTH_User starts her client program and selects the Expose function. She should be able to specify the name of the VO to which she wants her files to be exposed. If the VO does not exist it should be created. Then she selects the files to be exposed by using an explorer that shows a directory tree structure. In this case she selects the “documents” folder. Note that the following description includes some details of possible implementation that can be ignored at this stage of the VOFS design.

The following steps should be taken by KTH_User:

1. KTH_User searches for VO KTH_ICCS:
   a) KTH_Users queries the Resource (VO) Discovery Service (provided that such a service is available) for information on the VO named “KTH_ICCS”.
   b) The VO Discovery Service replies with information that includes a “FileSystem Resource Authority” property, which is, actually, a URI/URL is pointing to the MDDB serving the “KTH_ICCS” filesystem.

2. If there exists no VO called “KTH_ICCS”, it is created:
   a) A new DFS User, KTH_ICCS, is created, probably by the “VO Management” (or VOMAN for short).
   b) A new MDDB server is started (or an existing is used) to serve the KTH_ICCS filesystem (there is exactly one filesystem per DFS User). This should be arranged by the VOMAN as all other VO Resource Management issues (such as where the VO Membership Service executes).
   c) As the VOMAN completely controls the KTH_ICCS filesystem, it grants access permissions to the KTH_ICCS VO members.
   d) The KTH_ICCS filesystem is empty at this time.

3. KTH_User exposes her local folder C:\projects\documents\ to the KTH_ICCS filesystem (which is the authoritative KTH_ICCS VO filesystem) with the same name “documents” under the filesystem root directory. There are several approaches to do this:

(i) Exposition via copy
   a) KTH_User copies her files data in a storage Provider and the filesystem structure and metadata in the KTH_ICCS MDDB server.
   b) The storage Provider can be a third party or can be just a local copy of data. If the actual data are to be exported see alternative (ii). Now anyone can access the filesystem KTH_ICCS/documents.
   c) KTH_User can decide whether and how to copy/synchronize back and forth his local and MDDB versions of 'documents', manually.

(ii) Exposition via export / mount-back
   a) KTH_User exports the 'documents' folder as in the previous case (i). The storage Provider can be herself, offering direct access to the files.
   b) The local folder “documents” is hidden from the OS FS hierarchy but remains accessible by the Provider server. The KTH_ICCS/documents directory is mounted on top of the local “documents” directory.
   c) Now even local access to “documents” is really remote and upon unmounting, the local (“real”) documents directory can be synchronized if any modifications have been made when it was exposed.

d) KTH_User can decide that this folder is really a shared one and configure her system so that an MDDB server always present, and the “documents” folder is mounted from the local MDDB. Then a link can be created under “KTH_ICCS/documents” that point to KTH_User/documents. At this point, the actual data can be anywhere in the network, including the local storage Provider.

(iii) Exposition via exporter / synchronizer

a) The 'documents' directory is exported as in (i), with the local copy being independently accessible.

b) Another daemon, the 'exporter' or 'synchronizer' monitors both the MDDB version at KTH_ICCS/documents and the local version and tries to propagate changes between each other, in some way.

Note that the second solution (ii) might be the most interesting of all, especially considering the (d.) step.

KTH_User also should be able to specify a quota value for this expose. This value indicates how much storage space other users will have when they mount this folder and create new objects (files) or modify and extend existing objects (files).

Quotas can easily be maintained as metadata in the MDDB. The real problem, though, is enforcing quotas. But if each user is using his own storage resources for hosting file data, then quota is really a local issue. Shared storage resource quota can be managed within the framework that makes them shared anyway, irrespective of what the MDDB says about it.

By exposing, the necessary metadata of the file system of KTH_User will be sent to MDDB (Meta data database). KTH_User should also run a server program on her machine to provide access to the files she has exposed. Now other members of the KTH_ICCS VO (including KTH_User) should be able to see and browse this exposed set of files in the VOFS file hierarchy.

As presented in the second solution above entitled “(ii) Exposition via export / mount-back", the MDDB, which serves KTH_ICCS, may either include the whole filesystem structure or just a link from KTH_ICCS/documents to KTH_User/documents. In the first case, KTH_User doesn't have to run an MDDB. In the second case, KTH_User doesn't have to mirror her filesystem within another users' (i.e. KTH_ICCS) MDDB. But note, a KTH_User can request KTH_ICCS's MDDB to host her own filesystem. Then, KTH_ICCS/documents can be a link to a separate filesystem KTH_User/documents, but practically both filesystems are served by the same machine. This eliminates the need to run an MDDB per user while it maintains the flexibility of linking into filesystems rather than copying them in order to include them elsewhere. Therefore we strongly propose the use of linking where possible.

ICCS_User also runs her client and selects the Mount function. As a parameter she should indicate which VO she wants to browse, or she should be able to get a list of existing VOs. This list can be fetch from the metadata server. After specifying VO name (in this case KTH_ICCS), she browses the directory structure of the exposed files. She should be able to see the document folder that is exposed by KTH_User and possibly other folders that are exposed by other members of the VO.

4. ICCS_User browses VOs.

This can be provided as a VO Management service. Nevertheless, one can create a well-known Super-VO, which will include in its filesystem links to all available VOs. Anyway, we assume that ICCS_User knows how to contact the MDDB for KTH_ICCS.

5. ICCS_User mounts KTH_ICCS/documents

The filesystem client stores the KTH_ICCS/documents URI and the MDDB contact. All subsequent file requests under the mountpoint will be translated to MDDB requests for files under KTH_ICCS/documents. This is done incrementally, if necessary.

For example, ICCS_User isses a request

RETR(/local_mountpoint/projects/dow.pdf),

which is translated to an MDDB request to KTH_ICCS:

RETR(KTH_ICCS/documents/projects/dow.pdf)

If the file exists, then its metadata (its inode) is returned. But assume that 'KTH_ICCS/documents' is a link. Then the KTH_ICCS MDDB will reply with the deepest inode it found within its database that existed along the requested path. This will be, of course, 'KTH_ICCS/documents'. Now the ICCS_User filesystem client will realize that this was not what it asked for. Then it will attempt to continue, checking if the returned inode is actually a link. As it is a link that points to 'KTH_User/documents', so the MDDB request becomes:

RETR(KTH_User/documents/projects/dow.pdf)

This procedure is repeated until the client has reached and obtained the desired inode. Then it can retrieve the file data. In this way, as described above, one can browse through file systems in a P2P way, just like a web surfer follows the private links of each www user's webpage.

Now ICCS_User can choose a drive letter that is not in use in her local file system (assume MS Windows), for example Z: and selects the mount operation. The client program should create a remote drive Z: which contains the “documents” folder and all other sub folders in it. After mount, ICCS_User should see the following folders on the Z drive:

Z:\documents\meetings\meeting1.txt

Z:\documents\pictures\pic1.jpg

Z:\documents\maps\map1.pdf

She should be able to open and work with files in those folders in the same way as with her local files. If she tries to create new files on drive Z:\, the available free storage space should be checked. This parameter was indicated by KTH_User when she exposed her files.

6. The KTH_ICCS namespace is presented to the local FS.

This is quite independent from the DFS layer. The local filesystem can retrieve and cache DFS files and data in a similar way other filesystems retrieve and cache files from disk or network places.

If ICCS_User wants to create a new file, she has to find her own storage in a p2p or arbitrary way -- only storage pointers are needed in the MDDB. Of course, a storage allocation layer should be present to simplify this task. This is actually what the VBS will do. Storage quotas can be enforced in this level. Note that can stop anyone to create an impossibly large file if he himself provides the storage unless there is an explicit rule within the MDDB to only allow storage pointers of a certain origin.

From now on ICCS_User should be able to work on these files with arbitrary relevant application.

7. Implementation Plan of a VOFS Prototype

KTH has developed a prototype implementation of the VOFS in MS Windows. We used Java as main language. The prototype is based on component model. The system is constructed using several software components implemented in Java.

The first version of the prototype uses all components developed in WP3. However, in the future versions we plan to use some components from WP2, including authentication server and authorization server.

7.1 VOFS Components

7.1.1 MetaData Server

The MetaData Server is the central repository of all VOFS meta-data. Such meta-data includes endpoint information of all joining nodes, the logical naming space of VOFS files, physical addresses of VOFS files (including node endpoint information and file absolute path) and the mapping of logical and physical naming space. MetaData Server in the same time also provides catalog and search services for both VOFS file and directory information: The file/directory information records and maintains all kinds of file/directory attributions (meta-data). Catalog service aims to providing necessary operations to build a tree-like hierarchical catalog of VOFS and traverse on the tree. Search service aims to enabling query to locate a specific file or directory in the catalog tree.

7.1.2 Authentication Server

The Authentication Server serves as a centralized security center responsible for validating VOFS Clients and granting / issuing token to Clients. Being successfully authenticated by signing digital signature, clients hold certificated token to access other nodes in the system. The Authentication Server thus stores all public keys of users (key-store) which users hold their private keys. Besides, this server also maintains all ACL information of VOFS files. A Client can check permission of a file operation (a specific operation on a specific file) by checking it in the Authentication Server.

7.1.3 Authorization Service

Many conventional network file systems provide only coarsely grained ACL mechanism. For example, read only / write only / full operation of specific file or directory. VOFS aims at providing more finely grained ACL which includes more permission levels and more flexible role control such as which VOs or Groups are permitted for what kinds of file operation or which are not. The main mechanism is introduced as follows.

Each VOFS Logical Name (LN) has an Access Control List (ACL) attached. The permission types of ACL granted to User, Group or VO upon such LN can be included as:

1. List (L): Permission to list all sub-entries under directory.
2. Attribute (A): Permission to see all file/directory attributes
3. Read (R): Permission to read the file
4. Write (W): Permission to write the file
5. Rename (N): Permission to rename this file/directory
6. Mkdir (M): Permission to create new directory under this directory
7. Full (F): Full permissions including all above
8. Prohibit (P): Not able to access the file/directory

Except for “Full” and “Prohibit”, each every permission has three positive list and negative list pairs for User, Group and VO respectively to define which entities are allowed to such permission or not. (“Full” has only positive lists and “Prohibit” has only negative lists for User, Group and VO.) Meanwhile, similar to UNIX file system, each VOFS file entry has positive and negative permission. For example, a file entry has positive permission “F” means the file entry is of Full control and “LAR” means the file entry is subject to only “List”, “Attribute” and “Read” permissions, and vice versa for negative permission. When the positive permission and negative permission of the file entry have been identified, the according permission lists will be checked to see which entities are subject to these permissions. A more detailed explanation of VOFS authorization is given as follow.

The authorization of VOFS happens at file operation level. Assume authentication is successful and the connection between client user Ua and Ub has been established. Every time when Ua requests a file operation (say, delete operation) on a file Fx at Ub, Ua needs to first provide its identity and Ub will check with Authentication Server for ACL of that file, ACL(Fx). In a logical view, the negative lists (user/group/VO) of ACL(Fx) will be scanned first and then the positive lists. Ua’s identity will finally be validated in ACL(Fx).

It is guaranteed that user side can never fake or manipulate ACL since the authentication always takes place between trusted parities: the Authentication Server and file holder clients. In fact, those clients can always have their ACL at local. The reason Authentication Server is involved is that those clients can store the ACL information at the server in case the VOFS client program is terminated and the ACL in memory lost.

In order to deploy authorization service at Authentication Server, there will be one more step to store ACL of LNs. The client needs to exposes its local file entries to MetaData Server, and in addition, he needes to provide ACLs of those entries at the same time if there is any. The storage of ACL is realized by communication between user and the Authentication Server. Client users are able to store their ACLs at Authentication Server database and modify them at any time later. Every time a client logs to VOFS, it loads the ACLs from Authentication Server into its memory so as to reduce network communication and improve performance.

There is another option that is to deploy Authorization service with naming service at MetaData Server. In this scheme, local file exposure and ACL will no longer be split into two connections to different servers. But in order to make MetaData Server properly loaded, it is better option to adopt the first approach and assign authorization service to Authorization Server. In the current design of VOFS, it is essential to decouple the authorization service from MetaData Server since the naming resolution service, although not expensive, can be very frequently invoked by numerous client users. In addition, if all database related services are deployed at MetaData Server, it could become bottle neck or a single point of failure because of uncertainty of server’s disconnection from network or crashing. But it is feasible to have replications of MetaData data base and Authentication data base on both servers to improve the availability.

It is inevitable that there will be lots of requests for ACL checking which makes the program slowly responsive. To improve, a leisure mode can be added to exposed file entries (LNs) to bypass authorization. For those less sensitive entries, VOFS simply sets a symbol and treats them as shared ones, either read-only or with full operations.

Authorization and authorization have many requirements of VOFS security mechanism. It is never too safe to say a system is completely secure and immune from malicious attacks. In addition, employing too heavy security mechanism could be very expensive. VOFS should choose a medium level of security, easy to implement and with reasonable overhead.

7.1.4 Mount Server

The VOFS network has several Mount Servers that are running on top of stable workstations always connecting to the network and providing mounting service for VOFS client nodes. Before VOFS clients perform disconnection from VOFS, they can proactively upload their shared files/directories to MountServer

so that other client nodes can still access those file entries in a native way. More precisely, the accessing to those file entries is realized through native mounting of client nodes, as the mounting server is based on WebDAV which is a protocol natively supported by Linux similar OS as well as MS Windows. In some sense, the Mount Server is similar to file servers of AFS, but a simplified version with other administration features decoupled from it.

7.1.5 VOFS Client

The VOFS Client is defined as program installed at user computers. The VOFS Client program consists of two parts: the client interface and the file server. Client interface enables VOFS users to perform operations to expose their local file system, access remote files and call related services. The file server, on the other hand, provides service for remote access to local exposed file system.

Client Interface

Client Interface has layered structure. At the bottom is the communication layer responsible for sending messages and receiving responses. This layer is based on java network package and implemented with Java Socket. The communication is through message exchange in plain text.

Client Interface API is the core part which encapsulates user logics to operation all services.

The Client API includes the following functions:

1. Authentication service to log in user.
2. Local file system exposure to VOFS LN space.
3. Trigger VOFS LN resolution to locate PN.
4. Request catalog service of VOFS LN tree / search LN in the tree.
5. POSIX operations of remote file.
6. Asynchronous file transfer / block operation of file
7. Setting ACL for local exposed file system.
8. Mount remote directory to local.

On the top of Client Interface is Client UI of VOFS, including a GUI developed with Java Swing components and a command line tool. The core of GUI is VOFS explorer which includes two JTree based components: one built on native local file system and the other on VOFS LN hierarchy. Since different mouse events will be captured, the friendly GUI will help user locate any file entity in both hierarchical structures easily. The command line tool is more similar to UNIX style and user can input commands instead of clicking buttons to invoke Client APIs.

File Server

The file server in general is a user space daemon that receives file operation requests from other clients, checks authorization of the requests and executes the operations. Another feature is file-locking. When one remote client's is writing to local, the file server creates a lock for the file being written and other write or read requests will be blocked until the current write is done.

File server has a same layered structure as client interface, with a communication layer and file server API layer. File server API enables to invalidate the authorization of operation requests by checking with Authentication Server. It also enables all basic POSIX file operations at local file system which includes:

#	Name	Description
1	canRead	to check if the file can be read
2	canWrite	to check if the file can be written
3	createNewFile	to create a new file

4	delete	to delete a file or an empty folder
5	isFile	to check if it is a file
6	isDirectory	to check if it is a folder
7	Length	length of the file
8	Mkdirs	create hierarchy directories
9	List	list all files of the folder
10	renameTo	rename the file or folder
11	Exists	to check if the file or folder exists
12	Lock	to lock the file from accessing

Table 3: Basic POSIX file operations

8. References

[1] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, Wide-area cooperative storage with CFS, SOSP'01, 2001

[2] A. Adya, et al., Farsite: federated, available, reliable storage for an incompletely trusted environment, ACM SIGOPS Operating Systems Review, Vol.36, pp. 1—14, 2002.

[3] E. Laure, et al., Programming the Grid with gLite, EGEE-TR-2006-001, 2006. URL: http://doc.cern.ch/archive/electronic/egee/tr/egee-tr-2006-001.pdf

[4] EGEE Middleware Design Team. EGEE Deliverable 1.4: EGEE Middleware Architecture. https://edms.cern.ch/document/594698/

[5] A.-J. Peters, P. Saiz, and P. Buncic, AliEnFS - a Linux File System for the AliEn Grid Services, In Proc. Conf on Computing in High Energy and Nuclear Physics, 2003

[6] I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, Int'l J. High-Performance Computing Applications, vol. 15, no. 3, 2001, pp. 200-222.

[7] I. Foster, C. Kesselman, Scaling System-Level Science: Scientific Exploration and IT Implications, IEEE Computer, v. 39, no.11, Nov 2006, pp.31-39.

[8] A. Chervenak, E. Deelman, I. Foster, et al.., "Giggle: A Framework for Constructing Scalable Replica Location Servcies", In Proc. of Supercomputing'2002, 2002.

[9] W. Allcock, I. Foster, R. Madduri, "Reliable Data Transport: A Critical Service for the Grid", Building Service Based Grids Workshop, Global Grid Forum 11, June 2004.

[10] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe, "Wide Area Data Replication for Scientific Collaborations", in Proc. of 6th IEEE/ACM International Workshop on Grid Computing (Grid2005), 2005.

[11] LUFS: Linux Userland File System, URL: http://lufs.sourceforge.net/

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

[13] OpenAFS, URL: http://www.openafs.org/

[14] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A Security Architecture for Computational Grids, Proc. 5th ACM Conference on Computer and Communications Security Conference, pp. 83-92, 1998

[15] J. Zhang, and P. Honeyman, NFSv4 Replication for Grid Storage Middleware, Proc. of the 4th Int'l Workshop on Middleware for Grid Computing (MGC'06), Melbourne, Australia, ACM Press, 2006

[16] R.J. Figueiredo, N.H. Kapadia, J.A.B. Fortes, The PUNCH Virtual File System: Seamless Access To Decentralized Storage Services In A Computational Grid, Proc. of the 10th IEEE Int'l Symposium on High Performance Distributed Computing (HPDC'01), 2001

[17] S. Maad, et al., Towards a Complete Grid Filesystem Functionality – to appear in Future Generation Computer Systems, vol. 23, no. 1, 2007, 123–131.
URL: http://dx.doi.org/10.1016/j.future.2006.06.006

[18] F. García-Carballeira, et al., A Global and Parallel File System for Grids – to appear in Future Generation Computer Systems, vol. 23, no. 1, 2007, pp. 116-122,
URL: http://dx.doi.org/10.1016/j.future.2006.06.004

[19] R. Figueiredo, N. Kapadia and J. Fortes, "The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid", In Proc. of HPDC-10, San Francisco, CA, August 2001.

[20] AliEn: ALICE Environment, URL: http://alien.cern.ch

[21] Grid Datafarm – Gfarm file system, URL: http://datafarm.apgrid.org/

[22] Osamu Tatebe, et al, Grid Datafarm Architecture for Petascale Data Intensive Computing. In Proc. of the 2nd IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGrid 2002), pp. 102–110, 2002.

[23] Osamu Tatebe, et. Al, Grid Data Farm for Petascale Data Intensive Computing, Techn. Report TR-2001-4, Electrotechnical Laboratory, 2001 URL: http://datafarm.apgrid.org/pdf/gfarm-ETL-TR2001-4.pdf

[24] Grid File System Working Group (GFS-WG), URL: http://phase.hpcc.jp/ggf/gfs-rg/

[25] A. Jagatheesan, The GGF Grid File System Architecture Workbook, Tech'l memo, OGF Grid File System Working Group, URL: http://www.ggf.org/documents/GFD.61.pdf

[26] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In Proceedings of the Summer USENIX Technical Conference, pages 119-130, Portland, OR (USA), June 1985.

[27] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. Computer, 21(2):23-36, 1988.

[28] John K. Ousterhout. The role of distributed state. In R. Rashid, editor, CMU Computer Science: A 25th Anniversary Perspective, pages 199-217. ACM Press, 1991.

[29] M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. Computer, 23(5):9-18, 20-21, 1990.

[30] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. ACM Transactions on Computer Systems (TOCS), 10(1):3-25, 1992.

[31] C. Thekkath, T. Mann, and E. Lee. Frangipani: a scalable distributed file system. In SOSP '97: Proceedings of the Sixteenth ACM Dymposium on Operating Systems Principles, pages 224-237, Saint Malo, France, 1997. ACM Press.

[32] R. Guy, J. Heidemann, W. Mak, T. Page, Jr., G. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In USENIX Conference Proceedings, pages 63-71, Anaheim, CA, June 1990. USENIX.

[33] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In USENIX Summer, pages 238-247, 1986.

[34] D. Rosenthal. Evolving the Vnode interface. In USENIX Summer, pages 107-118, 1990.

[35] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. ACM Transactions on Computer Systems (TOCS), 14(1):41-79, 1996.

[36] J. Hartman and J. Ousterhout. The Zebra striped network file system. ACM Transactions on Computer Systems (TOCS), 13(3):274-310, 1995.

[37] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In FAST'03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, pages 1-14, San Francisco, CA, 2003. USENIX.

[38] J. Kubiatowicz. Extracting guarantees from chaos. Communications of the ACM, 46(2):33-38, 2003.

[39] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In Proceedings of IEEE Workshop on Mobile Computing Systems & Applications, pages 2-7, Santa Cruz, California, December 8-9 1994.

[40] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, pages 172-182, Copper Mountain, Colorado, US, 1995. ACM Press.

[41] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications (Special Issue: Recent Advances In Service Overlay Networks), 22(1):41-53, January 2004.

[42] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based Erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute (ICSI), Berkley, CA, USA, 1995.

[43] A. Grimshaw, A. Natrajan, M. Humphrey, M. Lewis, A. Nguyen-Tuong, J. Karpovich, M. Morgan, , and A. Ferrari. From Legion to Avaki: The Persistence of Vision. In F. Berman, G. Fox, and A. Hey, editors, Grid Computing: Making the Global Infrastructure a Reality, pages 265-298. John Wiley & Sons, March 2003.

[44] B. White, M. Walker, M. Humphrey, and A. Grimshaw. LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. In Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing, page 59, Denver, Colorado, 2001. ACM Press.

[45] A. Grimshaw, M. Herrick, and A. Natrajan. Avaki data grid - secure transparent access to data. In Ahmar Abbas, editor, Grid Computing: A Practical Guide To Technology And Applications. Charles River Media, 2003.

[46] D.W. Chadwick and A. Otenko. The PERMIS X.509 role based privilege management infrastructure. Future Generation Computer Systems, 19(2):277-289, February 2003.

[47] R. Alfieri, R. Cecchini, V. Ciaschini, L. Dell Agnello, A . Frohner, A. Gianoli, K. Lorentey, and F. Spataro. VOMS, an Authorization System for Virtual Organizations

[48] Security Assertion Markup Language (SAML) V2.0 Technical Overview. Working Draft 10, 9 October 2006. URL: http://www.oasis-open.org/committees/documents.php?wg_abbrev=security

[49] Pastis: A scalable multi-writer peer-to-peer file system: http://regal.lip6.fr/projects/pastis/pastis_en.html

[50] Pastry: A substrate for peer-to-peer applications: http://research.microsoft.com/%7Eantr/Pastry/

9. Appendix A: Access rights in AFS

Access rights in the Andrew File System (AFS) are shown in the table below. Every directory in AFS has its own ACL list. A newly created directory inherits the ACL of its parent directory. Any changes to the parent's ACL will not alter ACLs of subdirectories. Access permission area grated to users or/and groups of users.

Related to	Denotation	Access permission (access right)
File	R	The right to read the contents of the file
	w	The right to write the contents of the file.
	k	The right to lock the file
Directory	L	The right to list the names of files in the directory.
	I	The right to insert a file in the directory, e.g. the right to create a new file or directory
	d	The right to delete a file in the directory
	a	The right to change the ACL list. It does not grant any other rights.

Aliases for some common access rights	Refers to
all	All rights, i.e. rlidwka
write	Standard rights for writing file, i.e. rlidwk
read	standard rights for reading a file, i.e. rl
none	No rights whatsoever

10. Appendix B: Use Cases

1. CreateNewVO	
Scenario Name	CreateNewVO
Actor(s)	Tom: SystemAdmin
Flow of events	Tom, the system administrator of VOFS, is asked to create a new VO called Alpha in VOFS. He fills in all necessary information of new VO including unique identifier of VO and other contact information such as group email address. Also, a name virtual directory for Alpha is created in VOFS logical directory tree. Its attributes such as if it can be modified by other VO(s) or user group should be set.

2. CreateVOAdmin / CreateFileUser	
Scenario Name	CreateVO Admin/ CreateFileUser
Actor(s)	Tom: SystemAdmin, John: VOAdmin, Mary: FileUser
Flow of events	Tom, after creating the new VO, creates VOAdmin account for John of VO Alpha. John logs in and now he has permission to create FileUser account of the VO Alpha he's managing. He creates a FileUser account for Mary. Mary, the VOFS end user, now logs in as member of Alpha and can perform operations in VOFS. A default logic folder under Alpha is automatically created for her by her username.

3. ExpandDirectoryTree	
Scenario Name	ExpandDirectoryTree
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	Alice logs into VOFS client and by default she sees only root of VOFS tree, say "/vo1" and all VO directory under root. She left clicks on expand icon besides a directory and the directory is expanded by one level deeper. All attached files and sub folders will be shown. There are two types of sub folders under VO directory. One is linked, meaning it targets to an external endpoint. The other is unlinked, meaning it still has not linked yet.

4. CreateDir / DeleteDir	
Scenario Name	CreateDirectory
Actor(s)	Tom: SystemAdmin, John: VOAdmin, Mary: FileUser
Flow of events	Tom is able to create virtual directory in VOFS tree at any place by using "ExpandDirectoryTree". At any directory he right clicks and

	chooses “create sub directory” in menu. A new form will pop with all attributes to fill. The owner attribute is set to Tom’s user name and can’t be changed. Only one attribute, the unique name of the directory should be provided in compulsory. Once he confirms the action, the new directory is created. The created directory is either unlinked, which means the external link and machine endpoint attributes are left empty, or linked, with those two attributes filled. Tom can also set other attributes such as access permission (by which VO(s) and by which user group), read only, or extra accessing code, etc. He can also delete any virtual folder in the tree. John is able to do the same thing, but only in the VO he is administrating by default. Mary, the VOFS end user, by default can only do the same thing in her own directory named after her username in the directory of VO Alpha.

5. ModifyDirectoryAttribute	
Scenario Name	ModifyDirectoryAttribute
Actor(s)	Tom: SystemAdmin, John: VOAdmin, Mary: FileUser
Flow of events	Tom is able to modify attributes of existent VOFS directory by right clicking the directory and choose “modify attributes” in menu. The attributes include directory name, target link, access permission (by which VO(s) and by which user group), read only or hidden and extra accessing code, etc. He can also delete any virtual folder in the tree. John is able to do the same thing, but only in the VO he is administrating by default. Mary, the VOFS end user, by default can only do the same thing in the directory named by her username in the directory of VO Alpha.

6. SetWorkingSpace	
Scenario Name	SetWorkingSpace
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	In VOFS client, Alice needs to set working space to store cached replications of files exposed by other FileUsers. The directory of VOFS corresponds to the root “/vo1” of virtual directory tree by default. She needs to manually create a folder in her local file system and explicitly appoint this folder as working space in VOFS client.

7. ExposeLocalDirectory	
Scenario Name	ExposeLocalDirectory
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	In VOFS client explorer, Alice expands virtual directory tree and

	locates a virtual directory which has not been targeted to any external link yet (unlinked). She right clicks on a directory icon and menu appears. If option "expose here" is enabled, it mean she is able to expose her local directory / driver here. Otherwise the option is disabled and grey in colour. Alice chooses to expose her local directory to that point. By default this will trigger "ModifyDirectoryAttribute". The local directory path and physical machine net address will be filled automatically in this case. She can set the rest attributes. After she confirms the action, sub-entries of the exposed local folder will be listed under the virtual directory she chooses. The names of the entries are same as local while their attributes inherit from the virtual directory. By refreshing the VOFS client explorer, all FileUsers are able to view this change.

8. UnexposeLocalDirectory	
Scenario Name	ExposeLocalDirectory
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	In VOFS client explorer, Alice is able to view all the exposed entries in her local file system. She selects one exposed directory and right clicks "Unexpose". After confirmation, the local directory and its sub-entries are no longer exposed to VOFS.

9. ExposedDirectoryChange	
Scenario Name	ExposedDirectoryChange
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	If Alice creates or saves some new entries into exposed directory, the action will not lead to auto-exposing of those entries. She needs to do "ExposeLocalDirectory" again to expose those new entries. If Alice removes entries from directory, the system will automatically do "UnexposeLocalDirectory" of those entries.

10. GetExposedEntity	
Scenario Name	GetExposedEntity
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	Based on "SetWorkingspace", Alice expands virtual directory tree and right clicks a linked directory or a sub-entry file. In the menu she chooses "download remote entry". The popped form will give options of the local saving path of the remote entry. Default is the logical name path of the entry. The popped form also provides option if the downloaded entry is in consistent mode, which means to keep consistent with other replications of this entry in the same mode. By default the

	downloaded entry is not in consistent mode. After Alice confirms the operation, the target entry is downloaded to local working space in the saving path.

11. KeepEntityConsistent	
Scenario Name	GetExposedEntity
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	Based on “GetExposedEntity”, Alice can optionally set downloaded entity in consistent mode in popped form. If a replication is in consistent mode, it binds to the master copy at the file source provider. As long as Alice is online and the file is not open, system can update replication based on any source changes at runtime. If the file is open, client will pop warning message to notify Alice to save the current file to a new name and get her a latest replication of the source.

12. EnableMount	
Scenario Name	EnableMount
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	Alice wishes to set a local directory as mountable to other VOFS users. She right clicks on the local directory and chooses “set mount”. Popped form will provide access user name and password. The defaults are blank. After she confirms, the folder is mountable. Alice uses laptop and decides to go offline. She wants other users of VOFS keep on using her files by means of mounting. In client UI, she connects to a mount server and uploads the directory and set it as mountable. She can also set access username and account for her directory. Next time when Alice reconnects, she can choose to synchronize her local folder with the one on mount server.

13. DisableMount	
Scenario Name	DisableMount
Actor(s)	Alice: FileUser / All Deriving Actors
Flow of events	Alice wishes to disable the mount directory she previously set. She right clicks on the mountable local directory and chooses “disable mount”. (If the folder is not mountable, this option is not seen.) After she confirms, the folder is no longer mountable.

14. GetMountDirectory	
Scenario Name	GetMountDirectory
Actor(s)	Bob: FileUser / All Deriving Actors

Flow of events	Bob needs to mount Alice's mount folder. He expands the VOFS directory tree and right clicks Alice's folder. If the folder is set as mountable by Alice already, he can choose "get mount". The popped form asks information about the access username and password and local path / drive Bob wants to mount to. After Bob confirms, he can access the remote directory in a native way. Also one entry will be added to Bob's UI about his mount information.

15. RemoveMountDirectory	
Scenario Name	GetMountDirectory
Actor(s)	Bob: FileUser / All Deriving Actors
Flow of events	Bob no longer needs to mount Alice's mount folder. He goes to mount information in his UI and deletes the entry of Alice's directory. The local path/drive is disconnected from the remote directory.

Project no. 034567

Grid4All

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

Deliverable 3.1: Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities - Appendix I

Due date of deliverable: June, 2008.

Actual submission date: June, 2008.

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

Organisation name of lead contractor for this deliverable: INRIA Regal

Revision: Submitted 2007-06-20

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)	

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 the appendix I of Deliverable 3.1 ”Requirements analysis, design and implementation plan of Grid4All data storage and sharing facilities”, which comprises the following parts:

Chapter I	Semantic Store
Chapter II	Collaborative Applications
Chapter III	VO-File system
Appendix I	Telex application API

The following persons contributed to this appendix: Jean-Michel Busca INRIA-Regal, Marc Shapiro INRIA-Regal, Pierre Sutra INRIA-Regal.

Package
fr.inria.gforge.telex

Provides classes and interfaces of objects instanciated by Telex.

fr.inria.gforge.telex
Class ClosedDocumentException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.ClosedDocumentException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **ClosedDocumentException**
extends java.lang.Exception

Thrown when an application accesses a document that has been closed. The pathname of the document is provided in the detail message.

Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	ClosedDocumentException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

ClosedDocumentException

`public` **`ClosedDocumentException`**`(java.lang.String message)`

fr.inria.gforge.telex
Interface Document

public interface **Document**
extends

A shared document edited by a `TelexApplication`. This interface provides methods for viewing, updating and saving the document. These methods operate on the persistent state of the document, which comprises:

- A multi-log. It is the core data structure that represents the current state of the document. It contains a set of `Action`s, bound by `Constraint`s, that users submit to update to the document.
- A set of user-defined filters. each user may define its own view of the document, by applying `ActionFilter`s. Filters are named and stored as part of the document state, in a per-user name space.
- a set of per-user snapshots: each user may define `StateSnapshot` of the document that are of particular interest to him. Snapshots are named and stored stored as part of the document state, in a per-user name space.

When the document is open in `Telex.OpenMode.READ_ONLY` mode, only the methods that do not modify the persistent state of the document are allowed. When it is open in `Telex.OpenMode.READ_WRITE` or `Telex.OpenMode.CREATE` mode, all of the methods are allowed. Most of theses methods operate on behalf of the user who invoked the Virtual Machine, hereafter named *invoking user*.

Document updating. Invoking user may update the document by adding actions and constraints through the `addAction(Action)`, `addConstraint(Constraint)` and `addFragment(Fragment)` methods. Telex propagates these actions and constraints to peer Telex sites when connectivity permits, using a best-effort epidemic replication protocol. Updates to the document are periodically notified to the application through the `TelexApplication.execute(Document, ScheduleGenerator)` method.

Document viewing. Invoking user may assign a set of filters to the document through the `defineFilter(ActionFilter)` and the `removeFilter(ActionFilter)` methods. Filters are saved as part of the persistent state of the document, in the name space of invoking user. When invoking user opens a document, the set of filters that he has defined for this document is automatically restored and applied. This set may be obtained through the `listFilters()` method.

Document saving. Invoking user may define a set of document snapshots to retain through the `defineSnapshot(StateSnapshot)` and the `removeSnapshot(StateSnapshot)` methods. Snapshots are saved as part of the persistent state of the document, in the name space of invoking user. The set of snapshots currently defined by invoking user may be obtained through the `listSnapshots()` method.

Author:
J-M. Busca INRIA/Regal

Field Summary

`public static final`	`DEFAULT_TYPE` The type of a document whose name has no suffix. Value:

Method Summary

`void`	`addAction(Action action)` Adds the specified action to this document.
`void`	`addConstraint(Constraint constraint)` Adds the specified constraint to this document.
`void`	`addFragment(Fragment fragment)` Adds the specified multilog fragment to this document.

void	close() Closes this document.
boolean	defineFilter(ActionFilter filter) Adds the specified action filter to the set defined by invoking user for this document.
boolean	defineSnapshot(StateSnapshot snapshot) Adds the specified snasphot to the set defined by invoking user for this document.
boolean	executeNow(boolean force) Calls the execute() method on this document according to the specified mode.
void	garbageCollect(StateSnapshot snapshot) Garbage-collects the history of this document up to the specified snapshot.
TelexApplication	getApplication() Returns the application that is currently editing this document.
java.lang.String	getType() Returns the type of this document.
boolean	isClosed() Returns the open/closed status of this document.
boolean	isOffline() Returns the on-line/off-line status of this document.
ActionFilter[]	listFilters() Returns the set of action filters that are currently defined by invoking user for this document.
StateSnapshot[]	listSnapshots() Returns the set of snapshots that are currently defined by invoking user for this document.
boolean	removeFilter(ActionFilter filter) Removes the specified action filter from the set defined by invoking user for this document.
boolean	removeSnapshot(StateSnapshot snapshot) Removes the specified snaspshot from the set defined by invoking user for this document.
void	setSchedulingParameters(SchedulingParameters parameters) Applies the specified scheduling parameters to this document.
void	startFrom(Schedule schedule) Requests Telex to start from the specified schedule when generating new schedules.
void	voteFor(Schedule schedule) Votes for the specified schedule of actions.

Fields

DEFAULT_TYPE

```
public static final java.lang.String DEFAULT_TYPE
```

The type of a document whose name has no suffix.
Constant value:

(continued on next page)

(continued from last page)

Methods

getType

```
public java.lang.String getType()
```

Returns the type of this document. The type of a document is the suffix of its name, or DEFAULT_TYPE if the name has no suffix.

Returns:
the string representing the type of this document, possibly DEFAUL_TYPE.

getApplication

```
public TelexApplication getApplication()
```

Returns the application that is currently editing this document. This method returns null if this document was automatically opened because it is bound to another document.

Returns:
the the application that is currently editing this document, or null if there is no such application.

isOffline

```
public boolean isOffline()
```

Returns the on-line/off-line status of this document.

Returns:
`true` if this document is closed or off-line, and `false` otherwise.

isClosed

```
public boolean isClosed()
```

Returns the open/closed status of this document.

Returns:
`true` if this document is closed, and `false` otherwise.

close

```
public void close()
  throws java.io.IOException
```

Closes this document. After this call returns, the application will not be allowed any action on this document, and it will stop being notified of new events regarding this document.

Throws:
`IOException` - if an I/O error occurs.

addAction

```
public void addAction(Action action)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         InvalidFragmentException,
         java.io.IOException
```

(continued on next page)

(continued from last page)

Adds the specified action to this document. This is a convenience method which is semantically equivalent to:

```
Fragment fragment = new Fragment();
fragment.add(action);
addFragment(fragment);
```

See `addFragment(Fragment)` for more details.

Parameters:

`action` - the action to add.

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`InvalidFragmentException` - if the specified action is invalid.
`IOException` - if an I/O error occurs.

addConstraint

```
public void addConstraint(Constraint constraint)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         InvalidFragmentException,
         java.io.IOException
```

Adds the specified constraint to this document. This is a convenience method which is semantically equivalent to:

```
Fragment fragment = new Fragment();
fragment.add(constraint);
addFragment(fragment);
```

See `addFragment(Fragment)` for more details.

Parameters:

`constraint` - the constraint to add.

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`InvalidFragmentException` - if the specified constraint is invalid.
`IOException` - if an I/O error occurs.

addFragment

```
public void addFragment(Fragment fragment)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         InvalidFragmentException,
         java.io.IOException
```

(continued on next page)

(continued from last page)

Adds the specified multilog fragment to this document. This document must be open in read-write mode. The specified fragment must be valid, which means that (i) none of the specified actions must be associated with a document yet, (ii) all of the specified constraints must bind actions that are either in the specified fragment, or already associated with a document. When this method returns successfully, the document and timestamp attributes of all of the ctions contained in the specified fragment are set.

Parameters:

`fragment` - the fragment to add.

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`InvalidFragmentException` - if the specified fragment is invalid.
`IOException` - if an I/O error occurs.

---

executeNow

```
public boolean executeNow(boolean force)
```

Calls the `execute()` method on this document according to the specified mode. Re-generation of sound schedules is not necessary if (i) no action or constraint has been added to the document (ii) the set of active filters has not changed since the last time the execute() method was called. Nonetheless, the execution of the method can be forced in this case by setting the parameter of this method to true. If the execute() method is actually called, it excutes within the current (invoking) thread, and completes before this method returns.

Parameters:

`force` - the execute() method is called unconditionally when true, or only if necessary when false.

Returns:

true if the execute() has been called, and false otherwise.

---

startFrom

```
public void startFrom(Schedule schedule)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         InvalidScheduleException
```

Requests Telex to start from the specified schedule when generating new schedules. The request takes effect for schedules provided through the calls to the `execute()` that are made after this method returns.

Parameters:

`schedule` - the schedule to start from.

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`InvalidScheduleException` - if the specified schedule does not relate to this document.

---

voteFor

```
public void voteFor(Schedule schedule)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         InvalidScheduleException,
         java.io.IOException
```

Votes for the specified schedule of actions. [...]

Parameters:

`schedule` - the schedule to vote for.

(continued on next page)

(continued from last page)

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`InvalidScheduleException` - if the specified schedule is not legal.
`IOException` - if an I/O error occurs.

setSchedulingParameters

```
public void setSchedulingParameters(SchedulingParameters parameters)
```

Applies the specified scheduling parameters to this document. [...]

Parameters:

`parameters` - the scheduling parameters to apply.

defineFilter

```
public boolean defineFilter(ActionFilter filter)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         java.io.IOException
```

Adds the specified action filter to the set defined by invoking user for this document. If this method returns successfully, the specified filter is permanently saved as part of the persistent state of this document. This method itself does not force immediate re-generation of sound schedules: call the `executeNow(boolean)` method to do so.

Parameters:

`filter` - the filter to define.

Returns:

true if the filter is actually added, and false if the filter already existed for invoking user.

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`IOException` - if an I/O error occurs.

removeFilter

```
public boolean removeFilter(ActionFilter filter)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         java.io.IOException
```

Removes the specified action filter from the set defined by invoking user for this document. If the method returns successfully, the specified filter is permanently removed from the persistent state of this document. This method itself does not force immediate regeneration of sound schedules: call the `executeNow(boolean)` method to do so.

Parameters:

`filter` - the filter to remove.

Returns:

true if the filter is actually removed, and false if the filter did not exist for invoking user.

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`IOException` - if an I/O error occurs.

listFilters

```
public ActionFilter[] listFilters()
  throws ClosedDocumentException,
         java.io.IOException
```

Returns the set of action filters that are currently defined by invoking user for this document.

Returns:
the current set of action filters.

Throws:
`ClosedDocumentException` - if this document is closed.
`IOException` - if an I/O error occurs.

defineSnapshot

```
public boolean defineSnapshot(StateSnapshot snapshot)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         java.io.IOException
```

Adds the specified snasphot to the set defined by invoking user for this document. If this method returns successfully, the specified snapshot is permanently saved as part of the persistent state of this document.

Parameters:
`snapshot` - the snasphot to define.

Returns:
true if the snapshot is actually saved under its name, and false if a snapshot with the same name already existed for invoking user.

Throws:
`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`IOException` - if an I/O error occurs.

removeSnapshot

```
public boolean removeSnapshot(StateSnapshot snapshot)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         java.io.IOException
```

Removes the specified snaspshot from the set defined by invoking user for this document. If the method returns successfully, the specified snapshot is permanently deleted from the persistent state of this document.

Parameters:
`snapshot` - the snasphot to remove.

Returns:
true if the snapshot is actually removed, and false if the snapshot was not saved for invoking user.

Throws:
`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`IOException` - if an I/O error occurs.

(continued on next page)

(continued from last page)

listSnapshots

```
public StateSnapshot[] listSnapshots()
  throws ClosedDocumentException
```

Returns the set of snapshots that are currently defined by invoking user for this document.

Returns:

the current set of snasphots.

Throws:

`ClosedDocumentException` - if this document is closed.

---

garbageCollect

```
public void garbageCollect(StateSnapshot snapshot)
  throws ClosedDocumentException,
         IncompatibleOpenModeException,
         java.io.FileNotFoundException,
         InvalidDocumentStateException,
         java.io.IOException
```

Garbage-collects the history of this document up to the specified snapshot. The specified snapshot must be in the current set of snapshots defined by invoking user. It must be also materialized and stable.

Parameters:

`snapshot` - the state up to which garbage-collect this document.

Throws:

`ClosedDocumentException` - if this document is closed.
`IncompatibleOpenModeException` - if this document is opened in read-only mode.
`FileNotFoundException` - if the specified state is not found.
`InvalidDocumentStateException` - if the specified state is not stable.
`IOException` - if an I/O error occurs.

fr.inria.gforge.telex
Class IncompatibleOpenModeException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.IncompatibleOpenModeException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **IncompatibleOpenModeException**
extends java.lang.Exception

Thrown when a application writes to a document opened in read-only mode. The pathname of the document is provided in the detail message.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	IncompatibleOpenModeException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

IncompatibleOpenModeException

`public` **`IncompatibleOpenModeException`**`(java.lang.String message)`

fr.inria.gforge.telex
Class InvalidDocumentFormatException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.InvalidDocumentFormatException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **InvalidDocumentFormatException**
extends java.lang.Exception

Thrown when an application accesses a document whose internal structure is not valid. The pathname of the document is provided in the detail message.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	InvalidDocumentFormatException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

InvalidDocumentFormatException

`public` **`InvalidDocumentFormatException`**`(java.lang.String message)`

fr.inria.gforge.telex
Class InvalidDocumentStateException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.InvalidDocumentStateException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **InvalidDocumentStateException**
extends java.lang.Exception

Thrown when an application garbage-collects a document up to a state that is not valid. The name of the state is provided in the detail message.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	InvalidDocumentStateException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

InvalidDocumentStateException

`public` **`InvalidDocumentStateException`**`(java.lang.String message)`

fr.inria.gforge.telex
Class InvalidFragmentException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.InvalidFragmentException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **InvalidFragmentException**
extends java.lang.Exception

Thrown when an application passes as parameter a fragment that is not valid. The id of the fragment is provided in the detail message.

Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	InvalidFragmentException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

InvalidFragmentException

`public` **`InvalidFragmentException`**`(java.lang.String message)`

fr.inria.gforge.telex
Class InvalidScheduleException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.InvalidScheduleException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **InvalidScheduleException**
extends java.lang.Exception

Thrown when an application passes as parameter a schedule that is not valid. The id of the schedule is provided in the detail message.

Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	InvalidScheduleException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

InvalidScheduleException

`public` **`InvalidScheduleException`**`(java.lang.String message)`

fr.inria.gforge.telex
Interface Schedule

public interface **Schedule**
extends

A schedule of `Action`s to be applied on a Telex `Document`. It is defined by the sequence of actions to apply and the `DocumentState` to start from. A schedule is identified by an id of type String assigned by Telex when generating the schedule. A schedule only contains actions that belong to the document it relates to.

Author:
J-M. Busca INRIA/Regal

Method Summary

`Action[]`	`getActions()` Returns the sequence of actions corresponding to this schedule.
`Document`	`getDocument()` Returns the document this schedule relates to.
`java.lang.String`	`getId()` Returns the unique Id of this schedule.
`Action[]`	`getNonActions()` Returns the set of action that are excluded from this shedule.
`Schedule[]`	`getSchedules()` Returns the list of schedules that this schedule corresponds to in the case of bound documents.
`DocumentState`	`getState()` Returns the document state this schedule must be applied on.

Methods

getState

`public DocumentState getState()`

Returns the document state this schedule must be applied on.

Returns:
the document state this schedule must be applied on.

getActions

`public Action[] getActions()`

Returns the sequence of actions corresponding to this schedule.

Returns:
the sequence of actions corresponding to this schedule.

getNonActions

`public Action[] getNonActions()`

Returns the set of action that are excluded from this shedule. These action are excluded because they conflict with actions that belong to this schedule. The set of non actions is computed on demand.

Returns:
the set of action that are excluded from this shedule.

getId

`public java.lang.String getId()`

Returns the unique Id of this schedule.

Returns:
the unique Id of this schedule.

getDocument

`public Document getDocument()`

Returns the document this schedule relates to.

Returns:
the document this schedule relates to.

getSchedules

`public Schedule[] getSchedules()`

Returns the list of schedules that this schedule corresponds to in the case of bound documents.

Returns:
the list of schedules that this schedule corresponds to, or null if this schedule does not.

fr.inria.gforge.telex
Class Telex

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.Telex
```

public abstract class **Telex**
extends java.lang.Object

The main class of the Telex middleware. Several `TelexApplication` may run concurrently within a single Virtual Machine and use the services of Telex. Each application must first create a instance of Telex associated with it. Using this instance, the application may then open Telex `Document`s.

An application may process documents of various *types*. The type of a document is defined by the suffix of its name, e.g. "tdoc". Telex allows an application to associate `ProcessingParameters` with each of the document types it handles. When opening a document, Telex will apply to the document the parameters corresponding to its type. Note that the same document type may be registered by several applications in their respective Telex instance.

Each application may open one or more documents. The number of documents that can be opened is only limited by system ressources, mainly memory. A document that already exists may be open in either `Telex.OpenMode.READ_ONLY` or `Telex.OpenMode.READ_WRITE` mode. A new document may be created by opening it in `Telex.OpenMode.CREATE` mode. All opened documents are automatically closed when the Virtual Machine exits.

All applications running in a Virtual Machine execute on behalf of the user who invoked the VM. All the actions and all the constraints that these applications create will be attributed to this user. All Telex documents that these applications create will be owned by this user.

Depending on the value of the telex.mode java property, the Telex middleware may run in two modes:

- *Stand-alone* mode (telex.mode="standalone"). In this mode, Telex peers communicate through a P2P communication library. Each peer is responsible for keeping its own replica of the multi-log up-to-date. Telex manages only one document, and peers are known via static configuration files.
- *Grid4All* mode (telex.mode="grid4all"). In this mode, Telex relies on the underlying distributed file system for storing and replicating multi-logs, and for transmitting messages between Telex peers. Telex can handle several documents and the peers accessing a given document are determined dynamically.

Author:
J-M. Busca INRIA/Regal

Nested Class Summary

class	`Telex.OpenMode` Telex.OpenMode

Constructor Summary

protected	`Telex()`

Method Summary

static `Telex`	`getInstance(TelexApplication application)` Creates a Telex instance for the specified Telex application.

static Telex	getInstance(TelexApplication application, ProcessingParameters parameters) Creates a Telex instance for the specified Telex application and registers its default processing parameters.
static User	getInvokingUser() Returns the Id of the user who has invoked this Virtual Machine.
static boolean	isStandaloneMode() Indicates whether Telex is running in stand-alone mode in this Virtual Machine.
Document	openDocument(java.lang.String pathname) Opens the Telex document with the specified pathname in READ_WRITE mode.
abstract Document	openDocument(java.lang.String pathname, Telex.OpenMode mode) Opens the Telex document with the specified pathname in the specified mode.
abstract boolean	registerType(java.lang.String type, ProcessingParameters parameters) Registers the specified document type in this Telex instance and associates it with the specified processing parameters.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructors

Telex

protected **Telex**()

Methods

getInstance

public static Telex **getInstance**(TelexApplication application)

Creates a Telex instance for the specified Telex application. No document type is registered in this Telex instance after this method returns. The application must define at least one document type by calling the registerType(String, ProcessingParameters) method to be allowed to open documents.

Parameters:

application - the Telex application that registers.

Returns:

a Telex instance associated with application.

getInstance

public static Telex **getInstance**(TelexApplication application,
ProcessingParameters parameters)

(continued from last page)

Creates a Telex instance for the specified Telex application and registers its default processing parameters. This is a convenience method that is equivalent to:

```
Telex instance = Telex.getInstance(application);
instance.registerType(DEFAULT_TYPE, parameters);
```

Parameters:

`application` - the Telex application that registers.
`parameters` - the default processing parameters for application.

Returns:

a Telex instance associated with application.

getInvokingUser

`public static User getInvokingUser()`

Returns the Id of the user who has invoked this Virtual Machine. This user will be attributed all actions created within this VM and he will own all documents created within this VM.

Returns:

the Id of the user who has invoked this VM.

isStandaloneMode

`public static boolean isStandaloneMode()`

Indicates whether Telex is running in stand-alone mode in this Virtual Machine. If so, the behaviour of the following methods is affected: `openDocument()`, `Document.isOffline()`, [...]. See the description of these methods for more information.

Returns:

`true` if Telex runs in stand-alone mode, and `false` otherwise.

registerType

`public abstract boolean registerType(java.lang.String type, ProcessingParameters parameters)`

Registers the specified document type in this Telex instance and associates it with the specified processing parameters. If the specified type is `Document.DEFAULT_TYPE`, the specified parameters will be applied to documents whose type is not otherwise registered. A given document type can only be registered once: subsequent attempts to register it again will fail, as indicated by the value returned by this method.

Parameters:

`type` - the document type to register.
`parameters` - the processing parameters for type.

Returns:

true if the type was registered successfully, and false if the type was already registered.

(continued from last page)

openDocument

```
public Document openDocument(java.lang.String pathname)
  throws UnknownDocumentTypeException,
         java.io.FileNotFoundException,
         java.io.IOException,
         InvalidDocumentFormatException
```

Opens the Telex document with the specified pathname in `READ_WRITE` mode.

Parameters:

`pathname` - the pathname of the document.

Returns:

the corresponding `Document` object.

Throws:

`UnknownDocumentTypeException` - if the type of the document is not registered.
`FileNotFoundException` - if the document is not found.
`IOException` - if an I/O error occurs.
`InvalidDocumentFormatException` - if the internal structure of the document is incorrect.

openDocument

```
public abstract Document openDocument(java.lang.String pathname,
         Telex.OpenMode mode)
  throws UnknownDocumentTypeException,
         java.io.FileNotFoundException,
         java.io.IOException,
         InvalidDocumentFormatException
```

Opens the Telex document with the specified pathname in the specified mode. The type of the document, or the `Document.DEFAULT_TYPE` type, must be registered for the call to succeed. The specified document will be processed according to the parameters associated with its type.

Parameters:

`pathname` - the pathname of the document.
`mode` - the mode in which to open the document.

Returns:

the corresponding `Document` object.

Throws:

`UnknownDocumentTypeException` - if the type of the document is not registered.
`FileNotFoundException` - if the document is not found.
`IOException` - if an I/O error occurs.
`InvalidDocumentFormatException` - if the internal structure of the document is incorrect.

fr.inria.gforge.telex
Class Telex.OpenMode

```
java.lang.Object
   ┗-java.lang.Enum
      ┗-fr.inria.gforge.telex.Telex.OpenMode
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable

---

public static final class **Telex.OpenMode**
extends java.lang.Enum

The open mode of a Telex `Document`.

Field Summary

public static final	`CREATE` Create the specified document if it does not exist, and open it in read-write mode.
public static final	`READ_ONLY` Open the specified document in read-only mode.
public static final	`READ_WRITE` Open the specified document in read-write mode.

Method Summary

static `Telex.OpenMode`	`valueOf(java.lang.String name)`
static `Telex.OpenMode[]`	`values()`

Methods inherited from class `java.lang.Enum`

`clone, compareTo, equals, getDeclaringClass, hashCode, name, ordinal, toString, valueOf`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Methods inherited from interface `java.lang.Comparable`

`compareTo`

Fields

(continued from last page)

READ_ONLY

```
public static final fr.inria.gforge.telex.Telex.OpenMode READ_ONLY
```

Open the specified document in read-only mode.

READ_WRITE

```
public static final fr.inria.gforge.telex.Telex.OpenMode READ_WRITE
```

Open the specified document in read-write mode.

CREATE

```
public static final fr.inria.gforge.telex.Telex.OpenMode CREATE
```

Create the specified document if it does not exist, and open it in read-write mode.

Methods

values

```
public final static Telex.OpenMode[] values()
```

valueOf

```
public static Telex.OpenMode valueOf(java.lang.String name)
```

fr.inria.gforge.telex
Class UninstantiableClassException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.UninstantiableClassException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **UninstantiableClassException**
extends java.lang.Exception

Thrown when an application provides a parameter class that can not be instanciated. The name of the class is provided in the detail message.

Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	UninstantiableClassException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

UninstantiableClassException

`public` **`UninstantiableClassException`**`(java.lang.String message)`

fr.inria.gforge.telex
Class UnknownDocumentTypeException

```
java.lang.Object
   |-java.lang.Throwable
      |-java.lang.Exception
         |-fr.inria.gforge.telex.UnknownDocumentTypeException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **UnknownDocumentTypeException**
extends java.lang.Exception

Thrown when an application accesses a document whose type is not registered. The pathname of the document is provided in the detail message.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	UnknownDocumentTypeException(java.lang.String message)

Methods inherited from class `java.lang.Throwable`

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

UnknownDocumentTypeException

`public` **`UnknownDocumentTypeException`**`(java.lang.String message)`

fr.inria.gforge.telex
Class User

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.User
```

All Implemented Interfaces:
java.lang.Comparable, java.security.Principal

---

public class **User**
extends java.lang.Object
implements java.security.Principal, java.lang.Comparable

The description of a `TelexApplication` user. Each user is uniquely identified throughout the system by an id of type long.

This class has a *natural ordering* consistent with `equals()`. Moreover, objects of this class are canonical: no two distinct instances of this class may be equals().

Author:
J-M. Busca INRIA/Regal

Field Summary

public static final	`NOBODY` The *nobody* user.

Method Summary

int	`compareTo(User to)`
static User	`getInstance(long id)` Returns the description of the user with the specified id.
java.lang.String	`getName()`
java.lang.String	`toString()`

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Methods inherited from interface `java.security.Principal`

`equals, getName, hashCode, toString`

Methods inherited from interface `java.lang.Comparable`

`compareTo`

Fields

(continued from last page)

NOBODY

```
public static final fr.inria.gforge.telex.User NOBODY
```

The *nobody* user.

Methods

getInstance

```
public static User getInstance(long id)
```

Returns the description of the user with the specified id.

Parameters:
`id` - the id of the user to look up.

Returns:
the description of the user with the specified id.

---

toString

```
public java.lang.String toString()
```

---

getName

```
public java.lang.String getName()
```

---

compareTo

```
public int compareTo(User to)
```

Package
fr.inria.gforge.telex.application

Provides classes and interfaces of objects instanciated by Telex applications.

fr.inria.gforge.telex.application
Class Action

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.Action
```

public class **Action**
extends java.lang.Object

The generic representation of an action in Telex. As defined by the ACF, an action represent an operation on a `Document`. It has several generic attributes that applications may use as `ActionFilter` criteria. It also defines the application-specific keys attribute, which Telex uses to determine whether to call the appropriate `ConstraintChecker`. All of these attributes are immutable and they are all set when the action is created, except the document and timestamp attributes. These are set after a sucessfull call to `Document.addFragment(Fragment)`.

Unlike `Constraint`s, an action may belong to only one document. The triple (document, issuer, timestamp) uniquely identifies an action throughout the system. See the `Constraint` class on how constraints between actions are generated.

The *keys* of an action are a set of int values representing the application object that the action targets. Action keys are used to improve performance of constraint checking by avoiding unnecessary calls to the constraint checker. Given two actions of the same document, issued by distinct users, Telex calls the `ConstraintChecker.getConstraints(Action, Action)` method iif the key sets of the two actions intersect.

This class is meant to be sub-classed by Telex applications in order to define application-specific attributes such as the operation that the action represents, its parameters, etc. These attributes will be stored in the multi-log along the generic attributes and they will be transmitted to other peer Telex instances.

Author:
J-M. Busca INRIA/Regal

Field Summary

public static final	`INIT` The *INIT* special action as defined by the ACF.

Constructor Summary

public	`Action()` Creates a new action with no associated action key.
public	`Action(int[] keys)` Creates a new action associated with the specified keys.

Method Summary

`Document`	`getDocument()` Returns the Telex document this action belongs to.
`User`	`getIssuer()` Returns the user who created this action.
int[]	`getKeys()` Returns the keys associated with this action.
long	`getTime()` Returns the time this action was generated.

long	getTimestamp() Returns the timestamp of this action.
void	setDocument(DocumentImpl document) **Deprecated.**
void	setTimestamp(long timestamp) **Deprecated.**
java.lang.String	toString()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Fields

INIT

public static final fr.inria.gforge.telex.application.Action **INIT**

The *INIT* special action as defined by the ACF. By convention, this action is issued by user User.NOBODY at time 0. It has no associated key, it belongs to no specific document and it has an offset of 0.

Constructors

Action

public **Action**()

Creates a new action with no associated action key.

Action

public **Action**(int[] keys)

Creates a new action associated with the specified keys.

Methods

toString

public java.lang.String **toString**()

getKeys

public final int[] **getKeys**()

Returns the keys associated with this action. If this action is not associated with any action key, this method returns null. This attribute is set by Action constructors.

Returns:

(continued from last page)

the keys associated with this action, or null if no action key is associated with this action.

getIssuer

`public final User getIssuer()`

Returns the user who created this action. It is the user who invoked the Vitual Machine within which this action has been created. This attribute is set by Action constructors.

Returns:

the Id of the user who created this action.

getTime

`public final long getTime()`

Returns the time this action was generated. It is the time read on the system clock of the site that issued this action. This attribute should be considered with caution, since the system clocks of cooperating sites may not be accurate and/or synchronized. This attribute is set by Action constructors.

Returns:

the time this action was generated.

setDocument

`public final void setDocument(DocumentImpl document)`

Deprecated.

For Telex's internal use only.

getDocument

`public final Document getDocument()`

Returns the Telex document this action belongs to. This attribute is set after a successful call to the `Document.addFragment(Fragment)` with this action as parameter. Before such a call is made, this attribute is null.

Returns:

the Telex document this action belongs to, or null if no call to addFragment() with this action as parameter ever succeeded.

getTimestamp

`public final long getTimestamp()`

Returns the timestamp of this action. This attribute is set after a successful call to the `Document.addFragment(Fragment)` with this action as parameter. Before such a call is made, this attribute is -1.

Returns:

the timestamp of this action, or -1 if no call to addFragment() with this action as parameter ever succeeded.

setTimestamp

`public final void setTimestamp(long timestamp)`

Deprecated.

For Telex's internal use only.

fr.inria.gforge.telex.application
Class ActionFilter

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.ActionFilter
```

All Implemented Interfaces:
java.io.Serializable

Direct Known Subclasses:
RetainThisDocumentActionsOnly, ExcludeTheseUsersActions, RetainTheseUsersActionsOnly, RetainMyActionsOnly

---

public abstract class **ActionFilter**
extends java.lang.Object
implements java.io.Serializable

An object that specifies `Action`s to exclude from the view of a `Document`. An action filter may be associated with one or more documents by calling the `Document.defineFilter(ActionFilter)` method.

An action filter may be activated or de-activated by calling the `setActive(boolean)` method. The change of activity status applies to all documents that the filter is associated with. When generating sound `Schedule`s on the document, only the active filters of the document are applied. The activity status of an action filter is saved as part of the persistent state of each of the documents it is associated with when closing the document.

This class may be sub-classed by applications, for instance to provide methods for naming and/or describing a filter. A filter must be serializable in order to be saved as part of the persistent state of a document. This class provides several filters of general interest, as the `ActionFilter.RetainMyActionsOnly` filter.

Author:
P. Sutra INRIA/Regal, J-M. Busca INRIA/Regal

Nested Class Summary

class	`ActionFilter.ExcludeTheseUsersActions` ActionFilter.ExcludeTheseUsersActions
class	`ActionFilter.RetainMyActionsOnly` ActionFilter.RetainMyActionsOnly
class	`ActionFilter.RetainTheseUsersActionsOnly` ActionFilter.RetainTheseUsersActionsOnly
class	`ActionFilter.RetainThisDocumentActionsOnly` ActionFilter.RetainThisDocumentActionsOnly

Constructor Summary

public	`ActionFilter()` Creates an active action filter.
public	`ActionFilter(boolean active)` Creates an action filter with the specified activity status.

Method Summary

boolean	isActive() Determines whether this action filter is active.
abstract boolean	isFiltered(Action a) Determines whether the specified action must be filtered out.
void	setActive(boolean on) Sets the activity status of this action filter to the specified value.

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

ActionFilter

`public ActionFilter()`

Creates an active action filter.

ActionFilter

`public ActionFilter(boolean active)`

Creates an action filter with the specified activity status.

Parameters:
`active` - the activity status of the new action filter.

Methods

setActive

`public final void setActive(boolean on)`

Sets the activity status of this action filter to the specified value.

Parameters:
on - true is this filter must be activated, and false if this filter must be de-activated.

isActive

`public final boolean isActive()`

Determines whether this action filter is active.

Returns:
true is this action filter is active, and false otherwise.

isFiltered

`public abstract boolean isFiltered(Action a)`

(continued from last page)

Determines whether the specified action must be filtered out. This method is meant to be overriden by sub-classes: no default implementation is provided.

Parameters:

a - the action to check.

Returns:

true if the action must be filtered out, and false otherwise.

fr.inria.gforge.telex.application
Class ActionFilter.RetainMyActionsOnly

```
java.lang.Object
   |-fr.inria.gforge.telex.application.ActionFilter
      |-fr.inria.gforge.telex.application.ActionFilter.RetainMyActionsOnly
```

All Implemented Interfaces:
java.io.Serializable

---

public static class **ActionFilter.RetainMyActionsOnly**
extends ActionFilter

An action filter that retains only the actions that are issued by invoking user.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	ActionFilter.RetainMyActionsOnly() An action filter that retains only the actions that are issued by invoking user.

Method Summary

boolean	isFiltered(Action a)

Methods inherited from class fr.inria.gforge.telex.application.ActionFilter

isActive, isFiltered, setActive

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

---

Constructors

ActionFilter.RetainMyActionsOnly

```
public ActionFilter.RetainMyActionsOnly()
```

An action filter that retains only the actions that are issued by invoking user.

Methods

isFiltered

```
public final boolean isFiltered(Action a)
```

(continued from last page)

Determines whether the specified action must be filtered out. This method is meant to be overriden by sub-classes: no default implementation is provided.

fr.inria.gforge.telex.application
Class ActionFilter.RetainTheseUsersActionsOnly

```
java.lang.Object
   |-fr.inria.gforge.telex.application.ActionFilter
      |-fr.inria.gforge.telex.application.ActionFilter.RetainTheseUsersActionsOnly
```

All Implemented Interfaces:
java.io.Serializable

---

public static class **ActionFilter.RetainTheseUsersActionsOnly**
extends ActionFilter

An action filter that retains only the actions that are issued by a set of users.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	ActionFilter.RetainTheseUsersActionsOnly(java.util.Collection users) Creates an action filter that retains only the actions that are issued by the specified set of users.

Method Summary

boolean	isFiltered(Action a)

Methods inherited from class fr.inria.gforge.telex.application.ActionFilter

isActive, isFiltered, setActive

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

---

Constructors

ActionFilter.RetainTheseUsersActionsOnly

```
public ActionFilter.RetainTheseUsersActionsOnly(java.util.Collection users)
```

Creates an action filter that retains only the actions that are issued by the specified set of users.

Methods

isFiltered

```
public final boolean isFiltered(Action a)
```

(continued from last page)

Determines whether the specified action must be filtered out. This method is meant to be overriden by sub-classes: no default implementation is provided.

fr.inria.gforge.telex.application
Class ActionFilter.ExcludeTheseUsersActions

```
java.lang.Object
   |-fr.inria.gforge.telex.application.ActionFilter
      |-fr.inria.gforge.telex.application.ActionFilter.ExcludeTheseUsersActions
```

All Implemented Interfaces:
java.io.Serializable

---

public static class **ActionFilter.ExcludeTheseUsersActions**
extends ActionFilter

An action filter that excludes all the actions that are issued by a set of users.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	ActionFilter.ExcludeTheseUsersActions(java.util.Collection users) Creates an action filter that excludes all the actions that are issued by the specified set of users.

Method Summary

boolean	isFiltered(Action a)

Methods inherited from class fr.inria.gforge.telex.application.ActionFilter

isActive, isFiltered, setActive

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

---

Constructors

ActionFilter.ExcludeTheseUsersActions

```
public ActionFilter.ExcludeTheseUsersActions(java.util.Collection users)
```

Creates an action filter that excludes all the actions that are issued by the specified set of users.

Methods

isFiltered

```
public final boolean isFiltered(Action a)
```

(continued from last page)

Determines whether the specified action must be filtered out. This method is meant to be overriden by sub-classes: no default implementation is provided.

fr.inria.gforge.telex.application
Class ActionFilter.RetainThisDocumentActionsOnly

```
java.lang.Object
   |-fr.inria.gforge.telex.application.ActionFilter
      |-fr.inria.gforge.telex.application.ActionFilter.RetainThisDocumentActionsOnly
```

All Implemented Interfaces:
java.io.Serializable

---

public static class **ActionFilter.RetainThisDocumentActionsOnly**
extends ActionFilter

An action filter that retains only the actions belonging to a document.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	ActionFilter.RetainThisDocumentActionsOnly(Document document) Creates an action filter that retains only the actions belonging to the specified document.

Method Summary

boolean	isFiltered(Action a)

Methods inherited from class fr.inria.gforge.telex.application.ActionFilter

isActive, isFiltered, setActive

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

---

Constructors

ActionFilter.RetainThisDocumentActionsOnly

public **ActionFilter.RetainThisDocumentActionsOnly**(Document document)

Creates an action filter that retains only the actions belonging to the specified document.

Parameters:
document - the document whose actions must be retained only.

Methods

(continued from last page)

isFiltered

```
public final boolean isFiltered(Action a)
```

Determines whether the specified action must be filtered out. This method is meant to be overriden by sub-classes: no default implementation is provided.

fr.inria.gforge.telex.application
Class Constraint

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.Constraint
```

public class **Constraint**
extends java.lang.Object

The generic representation of a constraint in Telex. As defined by the ACF, a constraint binds two `Action`s and its `Constraint.Type` expresses the scheduling invariant between these actions. A constraint may reference the special `Action.INIT` action. INIT must then be the first action of the constraint and the type of the constraint must be `Constraint.Type.ENABLES`.

Most often, a constraint binds two actions of the same document: such a constraint is called an *intra-document* constraint. A constraint, however, may bind actions of two distinct documents: such a constraint is called a *cross-document* constraints, and the corresponding documents are said to be *bound*.

Intra-document constraints. These constraints may bind actions issued by either two distinct users or the same user. Telex automatically generates constraint of the former type by calling the `ConstraintChecker` associated with the document. By constrast, constraints of the latter type must be explicitely created and added to the document by calling one of the `Document.addFragment(Fragment)` method.

Cross-document constraints. These constraints must be explicitely created and added to both documents by calling one of the addFragment() methods, as in the following example:

```
// a1 and a2 are allocated actions, d1 and d2 are opened document
Constraint c = new Constraint(a1, ENABLES, a2);
d1.addAction(a1);
d2.addAction(a2);
d1.addConstraint(c);
d2.addConstraint(c);
```

A constraint generator may generate constraints in a deterministic fashion or not. A constraint is deterministically generated if the process of generating it only depends on the actions that the constraint binds. In particular, this process does not depend on the site where the constraint is generated, or on the state of the document when the constraint is generated. If a constraint is deterministically generated, Telex saves storage space and network bandwidth consumption by logging the constraint only once.

This class may be sub-classed by Telex applications in order to define application-specific attributes. These attributes will be stored in the multi-log along the generic attributes and they will be transmitted to other peer Telex instances.

Author:
J-M. Busca INRIA/Regal

Nested Class Summary

class	`Constraint.Type` Constraint.Type

Constructor Summary

public	`Constraint(Action first, Constraint.Type type, Action second)` Creates a constraint of the specified type between the specified actions.

Method Summary

Action	getFirstAction() Returns the first action that this constraint binds.
Action	getSecondAction() Returns the second action that this constraint binds.
Constraint.Type	getType() Returns the type of this constraint.
boolean	isDeterministic() Returns the generation mode of this constraint.
java.lang.String	toString()

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

Constraint

```
public Constraint(Action first,
                  Constraint.Type type,
                  Action second)
```

Creates a constraint of the specified type between the specified actions. If the type is ENABLES, the constraint specifies that the first action enables the second action. If the type is NOT_BEFORE, the constraint specifies that the first action is not scheduled before the second action.

Parameters:

`first` - the first action that this constraint binds.
`type` - the type of the constraint between first and second.
`second` - the second action that this constraint binds.

Methods

toString

```
public java.lang.String toString()
```

getType

```
public final Constraint.Type getType()
```

Returns the type of this constraint.

Returns:

the type of this constraint.

getFirstAction

`public final Action getFirstAction()`

Returns the first action that this constraint binds.

Returns:
the first action that this constraint binds.

getSecondAction

`public final Action getSecondAction()`

Returns the second action that this constraint binds.

Returns:
the second action that this constraint binds.

isDeterministic

`public boolean isDeterministic()`

Returns the generation mode of this constraint. Telex only checks this attribute on constraints issued by a constraint generator.

The default implementation of this method provided by this class always return true. It should be overriden by sub-classes if needed.

Returns:
true if this constraint is generated deterministically, and false otherwise.

fr.inria.gforge.telex.application
Class Constraint.Type

```
java.lang.Object
   |
   +-java.lang.Enum
      |
      +-fr.inria.gforge.telex.application.Constraint.Type
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable

---

public static final class **Constraint.Type**
extends java.lang.Enum

The type of an ACF constraint.

Field Summary

public static final	ENABLES The *enable* constraint.
public static final	NON_COMMUTING The *non-commuting* constraint.
public static final	NOT_AFTER The *not-after* constraint.

Method Summary

static Constraint.Type	valueOf(java.lang.String name)
static Constraint.Type[]	values()

Methods inherited from class java.lang.Enum

clone, compareTo, equals, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface java.lang.Comparable

compareTo

Fields

(continued from last page)

ENABLES

`public static final fr.inria.gforge.telex.application.Constraint.Type` **`ENABLES`**

The *enable* constraint.

NOT_AFTER

`public static final fr.inria.gforge.telex.application.Constraint.Type` **`NOT_AFTER`**

The *not-after* constraint.

NON_COMMUTING

`public static final fr.inria.gforge.telex.application.Constraint.Type` **`NON_COMMUTING`**

The *non-commuting* constraint.

Methods

values

`public final static Constraint.Type[]` **`values`**`()`

valueOf

`public static Constraint.Type` **`valueOf`**`(java.lang.String name)`

fr.inria.gforge.telex.application
Interface ConstraintChecker

All Known Implementing Classes:
CreationTimeOrder, NonCommutingActions

---

public interface **ConstraintChecker**
extends

An application-specific object that checks for Constraints between Actions. Telex associates with each Document the constraint checker that the application has registered for the type of the document, if any. Whenever an action a1 is added to the document, Telex checks for constraints between a1 and all existing action a2 that are not issued by the same user as a1. Telex performs this check whether action a1 is added by a local or a remote user.

Telex first tests whether actions a1 and a2 are likely to be bound by comparing their keys. If so, Telex then calls the getConstraints(Action, Action) method to get the actual set of contraints between a1 and a2. Finally, Telex adds the returned set, if any, to the document. See the the Constraint.isDeterministic() method for more details on how theses constraints are logged.

Note that Telex **never** checks for constraints between actions that are issued by the same user or that belong to two distinct documents, even if these documents are bound. See the Constraint class for more details on the classification of constraints.

This class provides two examples of constraint checkers.
Author:
P. Sutra INRIA/Regal, J-M. Busca INRIA/Regal

Nested Class Summary

class	ConstraintChecker.CreationTimeOrder ConstraintChecker.CreationTimeOrder
class	ConstraintChecker.NonCommutingActions ConstraintChecker.NonCommutingActions

Method Summary

Fragment	getConstraints(Action a, Action b) Determines the set of constraints between the specified actions.

Methods

getConstraints

```
public Fragment getConstraints(Action a,
                               Action b)
```

Determines the set of constraints between the specified actions. Telex calls this method only on actions that belongs to the same document and that are not isused by the same user.

Parameters:
a - an action.
b - another action.

Returns:

(continued from last page)

the set of constraints between action a and b, or null if there are no constraints.

fr.inria.gforge.telex.application
Class ConstraintChecker.NonCommutingActions

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.ConstraintChecker.NonCommutingActions
```

All Implemented Interfaces:
ConstraintChecker

---

public static final class **ConstraintChecker.NonCommutingActions**
extends java.lang.Object
implements ConstraintChecker

A constraint checker that specifies that no two actions commute.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	ConstraintChecker.NonCommutingActions() Creates constraint generator that specifies that no two actions commute.

Method Summary

Fragment	getConstraints(Action a, Action b)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface fr.inria.gforge.telex.application.ConstraintChecker

getConstraints

Constructors

ConstraintChecker.NonCommutingActions

```
public ConstraintChecker.NonCommutingActions()
```

Creates constraint generator that specifies that no two actions commute.

Methods

getConstraints

```
public Fragment getConstraints(Action a,
                               Action b)
```

fr.inria.gforge.telex.application
Class ConstraintChecker.CreationTimeOrder

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.ConstraintChecker.CreationTimeOrder
```

All Implemented Interfaces:
ConstraintChecker

---

public static final class **ConstraintChecker.CreationTimeOrder**
extends java.lang.Object
implements ConstraintChecker

A constraint checker that forces scheduling in creation-time order. This constraint checker relies on the system clock of cooperating sites, which may not be synchronized with one another.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	ConstraintChecker.CreationTimeOrder() Creates constraint generator that that forces scheduling in creation-time order.

Method Summary

Fragment	getConstraints(Action a, Action b)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface fr.inria.gforge.telex.application.ConstraintChecker

getConstraints

Constructors

ConstraintChecker.CreationTimeOrder

```
public ConstraintChecker.CreationTimeOrder()
```

Creates constraint generator that that forces scheduling in creation-time order.

Methods

getConstraints

```
public Fragment getConstraints(Action a,
                               Action b)
```

(continued from last page)

fr.inria.gforge.telex.application
Interface DocumentState

public interface **DocumentState**
extends java.io.Serializable

The state of a `Document` as maintained by a `TelexApplication`. This may be the Java object representing the document itself or any object that the application uses to identify such object. Document states are used when specifying `Schedule`s or materialized `StateSnapshot`s.

When saving a document state on persistent storage, Telex first splits it in fixed-size fragments. Telex then computes the SHA-1 hash of each fragment and saves the fragment under the (file) name representing this value. Consequently, fragments that remain unchanged from one snapshot of the state to another are reused instead of being stored twice, thus saving storage space. Moreover, fragments are stored in a storage space common to all users, thus enabling fragment sharing.

The application may help Telex to improve fragment re-use and sharing by specifying the appropriate fragments to use. This is the purpose of the `split()` and `assemble(Serializable[])` methods, which Telex calls before saving and after restoring the document state respectively.

The state of a document must be serializable in order to be saved on persistent storage.
Author:
J-M. Busca INRIA/Regal

Method Summary

void	`assemble(java.io.Serializable[] parts)` Provides the fragments that make up this document state.
java.io.Serializable[]	`split()` Returns the fragments that make up this document state.

Methods

split

`public java.io.Serializable[] split()`

Returns the fragments that make up this document state. Telex calls this method before saving this state on persistent storage.

Returns:
the fragments that make up this document state.

assemble

`public void assemble(java.io.Serializable[] parts)`

Provides the fragments that make up this document state. Telex calls this method after restoring this document state from persistent storage.

Parameters:
`parts` - the fragments that make up this state.

fr.inria.gforge.telex.application
Class Fragment

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.Fragment
```

public class **Fragment**
extends java.lang.Object

A set of Actions and Constraints considered as a whole.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	Fragment() Creates an empty fragment.
public	Fragment(java.util.Collection actions, java.util.Collection constraints) Creates a new fragment containing the specified sets of actions and constraints.
public	Fragment(Fragment fragment) Creates a new fragment containing the same elements as the given fragment.

Method Summary

boolean	add(Action action) Adds the specified action to this fragment.
boolean	add(Constraint constraint) Adds the specified constraint to this fragment.
boolean	add(Fragment fragment) Adds the elements of the specified fragment to this fragment.
void	clear() Removes all of the elements of this fragment.
java.util.Set	getActions() Returns the set of actions of this fragment.
java.util.Set	getConstraints() Returns the set of constraints of this fragment.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructors

(continued from last page)

Fragment

```
public Fragment()
```

Creates an empty fragment.

Fragment

```
public Fragment(java.util.Collection actions,
                java.util.Collection constraints)
```

Creates a new fragment containing the specified sets of actions and constraints. This is a convenience method which is equivalent to:

```
Fragment fragment = new Fragment();
for (Action a : actions) {
       fragment.add(a);
}
for (Constraint c : constraints) {
       fragment.add(c);
}
```

Parameters:

`actions` - the set of actions to initially place in this fragment, or null if the set is empty.
`constraints` - the set of constraints to initially place in this fragment, or null if the set is empty.

Fragment

```
public Fragment(Fragment fragment)
```

Creates a new fragment containing the same elements as the given fragment. The new fragment is created by using a copy-on-write mechanism.

Parameters:

`fragment` - the fragment whose elements are to be placed in this fragment.

Methods

getActions

```
public java.util.Set getActions()
```

Returns the set of actions of this fragment. The returned set is backed by this fragment, so changes to this fragment are reflected in the set, and vice-versa.

Returns:

the set of actions of this fragment.

getConstraints

```
public java.util.Set getConstraints()
```

(continued from last page)

Returns the set of constraints of this fragment. The returned set is backed by this fragment, so changes to this fragment are reflected in the set, and vice-versa.

Returns:
the set of constraints of this fragment.

---

add

```
public boolean add(Action action)
```

Adds the specified action to this fragment.

Parameters:
`action` - the action to add.

Returns:
true if this fragment did not already contain the specified action, and false otherwise.

---

add

```
public boolean add(Constraint constraint)
```

Adds the specified constraint to this fragment.

Parameters:
`constraint` - the constraint to add.

Returns:
true if this fragment did not already contain the specified constraint, and false otherwise.

---

add

```
public boolean add(Fragment fragment)
```

Adds the elements of the specified fragment to this fragment.

Parameters:
`fragment` - the fragment whose elements are to be added; null stands for the empty set.

Returns:
true if this fragment changed as a result of the call, and false otherwise.

---

clear

```
public void clear()
```

Removes all of the elements of this fragment. This fragment will be empty after this call returns.

fr.inria.gforge.telex.application
Class ProcessingParameters

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.ProcessingParameters
```

public class **ProcessingParameters**
extends java.lang.Object

The application-defined parameters for processing a `Document`. These fall into the followinf categories:

Constraint generation

Action scheduling

State storage

Functional extensions

These parameters comprise a `ConstraintChecker`, a set of default `SchedulingParameters`, a `ReplicaReconciler` and a `ScheduleGenerator`. The last two parameters are actually Java classes rather than Java object, as Telex needs to instanciate them.

Most `TelexApplication`s only have to define a specific constraint checker and possibly specific default scheduling parameters. Telex provides default implementations of the replica reconciler and the schedule generator that suit most of the needs. Only applications with very specific requirements need to provide their own replica reconciler and/or schedule generator.

Author:
J-M. Busca INRIA/Regal

Field Summary

public static final	`DEFAULT_PARAMETERS` The constant for "default processing parameters".

Constructor Summary

public	`ProcessingParameters()` Creates an instance of processing parameters with default values.
public	`ProcessingParameters(ConstraintChecker checker)` Creates an instance of processing parameters with the specified constraint checker.
public	`ProcessingParameters(ConstraintChecker checker, SchedulingParameters parameters)` Creates an instance of processing parameters with the specified constraint checker and default scheduling parameters.
public	`ProcessingParameters(ConstraintChecker checker, SchedulingParameters parameters, java.lang.Class reconciler)` Creates an instance of processing parameters with the specified constraint checker, default scheduling parameters and replica reconciler class.
public	`ProcessingParameters(ConstraintChecker checker, SchedulingParameters parameters, java.lang.Class reconciler, java.lang.Class scheduler)` Creates an instance of processing parameters with the specified constraint checker, default scheduling parameters, replica reconciler class and schedule generator class.

Method Summary

ConstraintChecker	getConstraintChecker() Returns the constraint checker defined by these parameters.
java.lang.Class	getReplicaReconciler() Returns the replica reconciler class defined by these parameters.
java.lang.Class	getScheduleGenerator() Returns the schedule generator class defined by these parameters.
SchedulingParameters	getSchedulingParameters() Returns the default scheduling parameters defined by these parameters.
ReplicaReconciler	instanciateReplicaReconciler() Returns a new instance of the replica reconciler class defined by these parameters.
ScheduleGenerator	instanciateScheduleGenerator() Returns a new instance of the schedule generator class defined by these parameters.
java.lang.String	toString()

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Fields

DEFAULT_PARAMETERS

```
public static final fr.inria.gforge.telex.application.ProcessingParameters
DEFAULT_PARAMETERS
```

The constant for "default processing parameters". This constant is created by calling the `ProcessingParameters()` constructor.

Constructors

ProcessingParameters

```
public ProcessingParameters()
```

Creates an instance of processing parameters with default values. These values are no constraint checker, `SchedulingParameters.DEFAULT_PARAMETERS`, `ReplicaReconciler.DEFAULT_RECONCILER` and `ScheduleGenerator.DEFAULT_SCHEDULER`.

ProcessingParameters

```
public ProcessingParameters(ConstraintChecker checker)
```

Creates an instance of processing parameters with the specified constraint checker.

Parameters:

`checker` - the constraint checker to select.

ProcessingParameters

```
public ProcessingParameters(ConstraintChecker checker,
                            SchedulingParameters parameters)
```

Creates an instance of processing parameters with the specified constraint checker and default scheduling parameters.

Parameters:

`checker` - the constraint checker to select.
`parameters` - the default scheduling parameters to select.

ProcessingParameters

```
public ProcessingParameters(ConstraintChecker checker,
                            SchedulingParameters parameters,
                            java.lang.Class reconciler)
```

Creates an instance of processing parameters with the specified constraint checker, default scheduling parameters and replica reconciler class.

Parameters:

`checker` - the constraint checker to select.
`parameters` - the default scheduling parameters to select.
`reconciler` - the class extending ReplicaReconciler to select.

Throws:

`UninstantiableClassException` - if the specified reconciler can not be instanciated.

ProcessingParameters

```
public ProcessingParameters(ConstraintChecker checker,
                            SchedulingParameters parameters,
                            java.lang.Class reconciler,
                            java.lang.Class scheduler)
```

Creates an instance of processing parameters with the specified constraint checker, default scheduling parameters, replica reconciler class and schedule generator class.

Parameters:

`checker` - the constraint checker to select.
`parameters` - the default scheduling parameters to select.
`reconciler` - the class extending ReplicaReconciler to select.
`scheduler` - the class extending ScheduleGenerator to select.

Throws:

`UninstantiableClassException` - if the specified reconciler or the specified scheduler can not be instanciated.

Methods

getConstraintChecker

```
public ConstraintChecker getConstraintChecker()
```

Returns the constraint checker defined by these parameters.

Returns:

the constraint checker defined by these parameters.

(continued from last page)

getSchedulingParameters

`public SchedulingParameters getSchedulingParameters()`

Returns the default scheduling parameters defined by these parameters.

Returns:
the default scheduling parameters defined by these parameters.

getReplicaReconciler

`public java.lang.Class getReplicaReconciler()`

Returns the replica reconciler class defined by these parameters.

Returns:
the replica reconciler class defined by these parameters.

getScheduleGenerator

`public java.lang.Class getScheduleGenerator()`

Returns the schedule generator class defined by these parameters.

Returns:
the schedule generator class defined by these parameters.

instanciateReplicaReconciler

`public ReplicaReconciler instanciateReplicaReconciler()`

Returns a new instance of the replica reconciler class defined by these parameters.

Returns:
a new instance of the replica reconciler class defined by these parameters.

instanciateScheduleGenerator

`public ScheduleGenerator instanciateScheduleGenerator()`

Returns a new instance of the schedule generator class defined by these parameters.

Returns:
a new instance of the schedule generator class defined by these parameters.

toString

`public java.lang.String toString()`

fr.inria.gforge.telex.application
Class SchedulingParameters

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.SchedulingParameters
```

public class **SchedulingParameters**
extends java.lang.Object

The action scheduling parameters of a `Document`. These parameters control how often Telex automatically calls the `TelexApplication.execute(Document, ScheduleGenerator)` method in response to new fragments being added to the document. Telex calls this method whenever one of the following conditions is true:

- the number of fragments added to the document since the last call exceeds a given value (threshold parameter),
- the time since a new fragment was added after the last call exceeds a given value (delay parameter).

When evaluating these conditions, Telex considers fragments added by both local site and remote sites, as well as calls to execute()

that the application may trigger unconditionally by calling the `Document.executeNow(boolean)` method.

Automatic calls to execute() may be de-activated by specifying the UNSPECIFIED_THRESHOLD and UNSPECIFIED_DELAY values for the threshold and delay parameters, respectively. In this case, the application may obtain new sound schedules only by calling executeNow().

When a document is opened, it is assigned the default scheduing parameters defined for its type. These parameters may be dynamically changed by calling the `Document.setSchedulingParameters(SchedulingParameters)`. When several documents are bound by cross-document constraints, Telex automatically set the scheduling parameters of all of the bound documents to the most restrictive values currently defined for these documents.

Author:
J-M. Busca INRIA/Regal

Field Summary

public static final	`DEFAULT_PARAMETERS` The constant for "default scheduling parameters".
public static final	`NO_DELAY` The constant for "no delay". Value: **0**
public static final	`NO_THRESHOLD` The constant for "no threshold". Value: **1**
public static final	`UNSPECIFIED_DELAY` The constant for "unspecified delay". Value: **-1**
public static final	`UNSPECIFIED_THRESHOLD` The constant for "unspecified threshold". Value: **0**

Constructor Summary

public	SchedulingParameters() Creates an instance of scheduling parameters with default values.
public	SchedulingParameters(int threshold, long delay) Creates an instance of scheduling parameters with the specified values.
public	SchedulingParameters(SchedulingParameters parameters) Creates an instance of scheduling parameters by duplicating the specified values.

Method Summary

long	getDelay() Returns the delay value of these scheduling parameters.
int	getThreshold() Returns the threshold value of these scheduling parameters.
void	merge(SchedulingParameters parameters) Merges the specified scheduling parameters into these scheduling parameters.
java.lang.String	toString()

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Fields

DEFAULT_PARAMETERS

`public static final fr.inria.gforge.telex.application.SchedulingParameters` **`DEFAULT_PARAMETERS`**

The constant for "default scheduling parameters". This constant is created by calling the `SchedulingParameters()` constructor.

NO_THRESHOLD

`public static final int` **`NO_THRESHOLD`**

The constant for "no threshold". If threshold is set to this value, sound schedules are computed every time a new fragment is added to the document, regardless of the delay value.
Constant value: **`1`**

UNSPECIFIED_THRESHOLD

`public static final int` **`UNSPECIFIED_THRESHOLD`**

The constant for "unspecified threshold". If threshold is set to this value, schedule computing is only governed by the delay parameter.
Constant value: **`0`**

NO_DELAY

`public static final int` **`NO_DELAY`**

(continued from last page)

The constant for "no delay". If delay is set to this value, sound schedules are computed every time a new fragment is added to the document, regardless of the threshold value.
Constant value: **0**

UNSPECIFIED_DELAY

```
public static final int UNSPECIFIED_DELAY
```

The constant for "unspecified delay". If delay is set to this value, schedule computing is only governed by the threshold parameter.
Constant value: **-1**

Constructors

SchedulingParameters

```
public SchedulingParameters()
```

Creates an instance of scheduling parameters with default values. The threshold parameter is set to NO_THRESHOLD, and the delay parameter is set to NO_DELAY.

SchedulingParameters

```
public SchedulingParameters(int threshold,
                            long delay)
```

Creates an instance of scheduling parameters with the specified values. if the threshold (resp. delay) value is less or equal to 0, it is silently set to NO_THRESHOLD (resp. NO_DELAY).

Parameters:

`threshold` - the value of the threshold parameter.
`delay` - the value of the delay parameter, in millisecond.

SchedulingParameters

```
public SchedulingParameters(SchedulingParameters parameters)
```

Creates an instance of scheduling parameters by duplicating the specified values.

Parameters:

`parameters` - the parameters to duplicate.

Methods

getThreshold

```
public int getThreshold()
```

Returns the threshold value of these scheduling parameters.

Returns:

the threshold value of these scheduling parameters.

getDelay

```
public long getDelay()
```

Returns the delay value of these scheduling parameters.

(continued from last page)

Returns:

the delay value of these scheduling parameters.

toString

```
public java.lang.String toString()
```

merge

```
public void merge(SchedulingParameters parameters)
```

Merges the specified scheduling parameters into these scheduling parameters. This method compares both parameters and retains the most restrictive, i.e. the smallest, value of each of the threshold and delay fields.

Parameters:

`parameters` - the scheduling parameters to merge into these ones.

fr.inria.gforge.telex.application
Class StateSnapshot

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.application.StateSnapshot
```

All Implemented Interfaces:
java.io.Serializable

---

public abstract class **StateSnapshot**
extends java.lang.Object
implements java.io.Serializable

A snapshot of the state of a `Document`. It is defined by the `Schedule` of actions producing the state that the snapshot represents. A snapshot is identified by a name, which Telex uses to save the snasphot to persistent storage when the `Document.defineSnapshot(StateSnapshot)` method is called. Telex also records the time when the snapshot is saved

A snapshot is said to be *materialized* if the corresponding `DocumentState` is provided with the snapshot. A snapshot is said to be *stable* if all of the actions of the corresponding schedule are stable. Materialized and stable snasphots are the only snapshots that may be considered as garbage-collection points.

This class may be sub-classed by applications, for instance to provide the description and the creation time of the snapshot. A snapshot must be serializable in order to be saved as part of the persistent state of the document.
Author:
J-M. Busca INRIA/Regal

Constructor Summary

public	`StateSnapshot(java.lang.String name, Schedule schedule)` Creates a simple snapshot with the specified name corresponding to the specified schedule.
public	`StateSnapshot(java.lang.String name, Schedule schedule, DocumentState state)` Creates a materialized snapshot with the specified name corresponding to the specified schedule and state.

Method Summary

boolean	`equals(java.lang.Object object)`
java.lang.String	`getName()` Returns the name of this snapshot.
Schedule	`getSchedule()` Returns the schedule of actions producing the state that this snasphot defines.
DocumentState	`getState()` Returns the state that this snapshot defines.
long	`getTime()` Returns the time this snasphot was created.
int	`hashCode()`

boolean	isMaterialized() Determines whether this snapshot is materialized.
boolean	isStable() Determines whether this snapshot is stable.
void	remove()
void	save()
void	setDocument(DocumentImpl document) **Deprecated.**

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

StateSnapshot

```
public StateSnapshot(java.lang.String name,
                     Schedule schedule)
```

Creates a simple snapshot with the specified name corresponding to the specified schedule.

StateSnapshot

```
public StateSnapshot(java.lang.String name,
                     Schedule schedule,
                     DocumentState state)
```

Creates a materialized snapshot with the specified name corresponding to the specified schedule and state. The constructor stores a copy of the specified state that it creates via serialization - de-serialization. The specified state must not be updated during the time this constructor executes.

Methods

setDocument

```
public final void setDocument(DocumentImpl document)
```

Deprecated.

Associate this snaspshot with the specified document. This methods checks that the snapshot do relate to the specified document. Then it saves this snasphot on persistent storage.

For Telex's internal use only.

getName

```
public final java.lang.String getName()
```

Returns the name of this snapshot.

(continued from last page)

Returns:
the name of this snapshot.

getTime

```
public final long getTime()
```

Returns the time this snasphot was created. This attribute is set by Telex when the `Document.defineSnapshot(StateSnapshot)` method is called on this snasphot.

Returns:
the time this snasphot was created.

getSchedule

```
public final Schedule getSchedule()
```

Returns the schedule of actions producing the state that this snasphot defines.

Returns:
the schedule of actions producing the state that this snasphot defines.

getState

```
public final DocumentState getState()
```

Returns the state that this snapshot defines. If this snapshot is not materialized, this method returns null.

Note that when a snapshot is retrieved from persistent storage, the corresponding state, if any, is not loaded into memory until this method is called. In order to know whether this snapshot is materialized, call the `isMaterialized()` method instead.

Returns:
the state that this snapshot defines, or null if this snapshot is not materialized.

hashCode

```
public int hashCode()
```

equals

```
public boolean equals(java.lang.Object object)
```

isMaterialized

```
public final boolean isMaterialized()
```

(continued from last page)
Determines whether this snapshot is materialized. This method is semantically equivalent to:

```
getState() != null
```

Nonetheless, this method should be prefered to `getState()` since calling the latter method actually load the state in memory.

Returns:
true if this snapshot is materialized, and false otherwise.

isStable

```
public final boolean isStable()
```

Determines whether this snapshot is stable.

Returns:
true if this snapshot is stable, and false otherwise.

save

```
public void save()
```

remove

```
public void remove()
```

fr.inria.gforge.telex.application
Interface TelexApplication

public interface **TelexApplication**
extends

An application that uses the services of `Telex`. This interface defines upcalls that Telex uses to notify the application of various events.

All these events relate to a specific `Document` that the application has opened by calling the `Telex.openDocument(String, Telex.OpenMode)` method. Telex never notifies the application of events regarding documents that the application has not opened explicitely, or that it has closed. In particular, the application is not notified of events regarding a document d2 that is bound to document d1 that the application has opened, unless the application explicitely opens d2.

Author:
P. Sutra INRIA/Regal, J-M. Busca INRIA/Regal

Method Summary

void	`bindDocument(Document opened, Document bound)` Notifies this application that the specified documents are bound.
void	`execute(Document document, Schedule schedule)` Request this application to execute the specified schedule on the specified document.
void	`execute(Document document, ScheduleGenerator generator)` Requests this application to trigger the specified schedule generator on the specified document and apply the generated schedule(s).

Methods

execute

```
public void execute(Document document,
         ScheduleGenerator generator)
```

Requests this application to trigger the specified schedule generator on the specified document and apply the generated schedule(s). Telex automatically calls this method whenever conditions for generating new schedules are met on the specified document. This method is also called as a result of this application calling the `Document.executeNow(boolean)` on the specified document.

Parameters:
`document` - the document generator relate to.
`generator` - the schedule generator for document.

bindDocument

```
public void bindDocument(Document opened,
         Document bound)
```

Notifies this application that the specified documents are bound. Telex calls this method when it discovers that document opened, currently opened by this application, is actually bound with document bound by a cross-document constraint. For each pair (opened, bound), Telex calls this method only once after document opened has been opened. Note that when this method is called, document bound may be opened or closed. If opened, the application that edits it may be this application or another application.

(continued from last page)

Parameters:

`opened` - a document currently opened by this application.

`bound` - the document that is bound to current.

---

execute

```
public void execute(Document document,
         Schedule schedule)
```

Request this application to execute the specified schedule on the specified document. This method is only called when the specified document is bound to another document bound, which is opened in read-write mode by some application app. Application app uses it to synchronize with this application in order to display related schedules on document and the specified document.

Parameters:

`document` - the document schedule relates to.

`schedule` - the schedule to apply on document.

Package
fr.inria.gforge.telex.extensions

Provides classes and interfaces for extending the core functionalities of Telex.

fr.inria.gforge.telex.extensions
Class ComponentIterableScheduleGenerator

```
java.lang.Object
   |-fr.inria.gforge.telex.extensions.ScheduleGenerator
      |-fr.inria.gforge.telex.extensions.ComponentIterableScheduleGenerator
```

public abstract class **ComponentIterableScheduleGenerator**
extends ScheduleGenerator

A connected component-aware iterable `ScheduleGenerator`.
Author:
J-M. Busca INRIA/Regal

Fields inherited from class `fr.inria.gforge.telex.extensions.ScheduleGenerator`
`_activeFilters`, `_graphSnapshot`, `_previousGenerator`, `_relatedDocuments`, `DEFAULT_SCHEDULER`

Constructor Summary

public	`ComponentIterableScheduleGenerator()`

Method Summary

abstract `Schedule`	`getCommittedSchedule()` Returns the schedule that has been committed so far.
abstract `Action[]`	`getDeadActions()` Returns the set of dead actions.
abstract `Action[]`	`getIndependentActions()` Returns the set of independent actions.
abstract `java.util.Iterator[]`	`getScheduleIterators()` Returns a set of iterators over sound schedules, one for each connected component of the action-constraint graph.

Methods inherited from class `fr.inria.gforge.telex.extensions.ScheduleGenerator`
`getInstance`, `isReleased`, `setParameters`

Methods inherited from class `java.lang.Object`
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

(continued from last page)

ComponentIterableScheduleGenerator

`public` **`ComponentIterableScheduleGenerator`**`()`

Methods

getDeadActions

`public abstract` `Action[]` **`getDeadActions`**`()`

Returns the set of dead actions.

Returns:
the set of dead actions.

---

getCommittedSchedule

`public abstract` `Schedule` **`getCommittedSchedule`**`()`

Returns the schedule that has been committed so far.

Returns:
the schedule that has been committed so far.

---

getIndependentActions

`public abstract` `Action[]` **`getIndependentActions`**`()`

Returns the set of independent actions.

Returns:
the set of independent actions.

---

getScheduleIterators

`public abstract java.util.Iterator[]` **`getScheduleIterators`**`()`

Returns a set of iterators over sound schedules, one for each connected component of the action-constraint graph.

Returns:
an set of iterator over sound schedules.

fr.inria.gforge.telex.extensions
Class IterableScheduleGenerator

```
java.lang.Object
   |-fr.inria.gforge.telex.extensions.ScheduleGenerator
      |-fr.inria.gforge.telex.extensions.IterableScheduleGenerator
```

All Implemented Interfaces:
java.lang.Iterable

---

public class **IterableScheduleGenerator**
extends ScheduleGenerator
implements java.lang.Iterable

A `ScheduleGenerator` that generates schedules upon request. Generating a sound schedule is CPU-consumming, and a large number of them may exist for a given action-constraint graph. It is therefore not possible to compute all sound schedules beforehand. Beside, the application may be interested only in a few or even just one of them. The purpose of this schedule generator is to save CPU power by generating sound schedule dynamically, upon application request. The application may iterate through the generated schedule and stop when satisfied. This schedule generator complies with the guidelines described in `ScheduleGenerator`.

Author:
J-M. Busca INRIA/Regal

Fields inherited from class `fr.inria.gforge.telex.extensions.ScheduleGenerator`
`_activeFilters`, `_graphSnapshot`, `_previousGenerator`, `_relatedDocuments`, `DEFAULT_SCHEDULER`

Constructor Summary

public	`IterableScheduleGenerator()` Creates a iterable schedule generator.

Method Summary

`Schedule`	`getCommittedSchedule()` Returns the schedule that has been committed so far.
`Action[]`	`getDeadActions()` Returns the set of dead actions.
`IterableScheduleGenerator`	`getInstance(DocumentImpl document)`
boolean	`isReleased()`
java.util.Iterator	`iterator()` Returns an iterator over sound schedules.
void	`release()` Releases this schedule generator.

Methods inherited from class `fr.inria.gforge.telex.extensions.ScheduleGenerator`
`getInstance`, `isReleased`, `setParameters`

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface java.lang.Iterable
iterator

Constructors

IterableScheduleGenerator

`public IterableScheduleGenerator()`

Creates a iterable schedule generator.

Methods

getInstance

`public IterableScheduleGenerator getInstance(DocumentImpl document)`

Returns a schedule generator instance responsible for the specified document. Telex calls this method for each document of the `ScheduleGenerator._relatedDocuments` list, even if it contains only one element. Telex passes the returned schedule generator to the execute() method.

isReleased

`public boolean isReleased()`

Determines whether this schedule generator has been released. Telex calls this method to check whether it can call the `TelexApplication.execute(Document, ScheduleGenerator)` method again for the same document.

getDeadActions

`public Action[] getDeadActions()`

Returns the set of dead actions.

Returns:
the set of dead actions.

getCommittedSchedule

`public Schedule getCommittedSchedule()`

Returns the schedule that has been committed so far.

Returns:
the schedule that has been committed so far.

iterator

`public java.util.Iterator iterator()`

(continued from last page)

Returns an iterator over sound schedules. Calling the `Iterator.hasNext()` method on the returned iterator actually compute a new sound schedule, if any. Calling the `Iterator.next()` method retrieves the schedule just computed. The `Iterator.remove()` method is not supported: a call to this method throws the UnsupporteOperationException.

Returns:

an iterator over sound schedules.

---

release

```
public void release()
```

Releases this schedule generator.

fr.inria.gforge.telex.extensions
Class ReplicaReconciler

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.extensions.ReplicaReconciler
```

Direct Known Subclasses:
VotingReplicaReconciler

---

public abstract class **ReplicaReconciler**
extends java.lang.Object

An object that reconciles `Schedule`s generated at different sites. Sites that cooperatively edit a `Document` may generate different sound schedules from the same set of actions and constraints. The purpose of the replica reconciler is to make cooperating sites agree on a common schedule and thus achieve (eventual) mutual consistency.

Telex provides a default general-purpose reconciler: `DEFAULT_RECONCILER`. A `TelexApplication` with specific needs may replace this default reconciler with its own by specifying it in its `ProcessingParameters`. The application-specific reconciler must extend the class `ReplicaReconciler` class and provide a public nullary constructor.

Telex creates a new reconciler object whenever a new `Document` is opened. The reconciler is then passed the document it is associated with. The reconciler must interface with the `fr.inria.gforge.telex.scheduling.Scheduler` associated with the document. On one hand, the scheduler notifies the reconciler of new fragment being added to the action-constraint graph it maintains. On the other hand, the reconciler materializes scheduling decisions as a set of constraints that it adds to the action-constraint graph. The reconciler may also interface with the `fr.inria.gforge.telex.communication.Transmitter` associated with the document in order to communicate with peer Telex instances that are currently editing this document.
Author:
J-M. Busca INRIA/Regal

Field Summary

protected	`_targetDocument`
public static final	`DEFAULT_RECONCILER` The default replica reconciler of Telex.

Constructor Summary

protected	`ReplicaReconciler()` Creates a replica reconciler.

Method Summary

Scheduler	`getScheduler()` Returns the scheduler associated with this reconciler.
Transmitter	`getTransmitter()` Returns the message transmitter associated with this reconciler.
abstract void	`graphUpdate()` Notifies this reconciler that the action-constraint graph was updated.
abstract void	`receiveVote(ScheduleImpl schedule)` Notifies this reconciler that a vote for the specified schedule has been received.

void	setParameters(DocumentImpl document) Sets the parameters of this replica reconciler.
abstract void	voteFor(ScheduleImpl schedule) Requests this reconciler to favor the specified schedule when making scheduling decisions.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Fields

DEFAULT_RECONCILER

public static final java.lang.Class **DEFAULT_RECONCILER**

The default replica reconciler of Telex. This constant is the VotingReplicaReconciler class.

_targetDocument

protected fr.inria.gforge.telex.implementation.DocumentImpl **_targetDocument**

Constructors

ReplicaReconciler

protected **ReplicaReconciler**()

Creates a replica reconciler.

Methods

setParameters

public final void **setParameters**(DocumentImpl document)

Sets the parameters of this replica reconciler. Telex calls this method just after creating this replica reconciler, and before any other method is called.

Parameters:
document - the document that this reconciler must handle.

getScheduler

protected final Scheduler **getScheduler**()

Returns the scheduler associated with this reconciler.

Returns:
the scheduler associated with this reconciler.

(continued from last page)

getTransmitter

```
protected final Transmitter getTransmitter()
```

Returns the message transmitter associated with this reconciler.

Returns:
the message transmitter associated with this reconciler.

---

graphUpdate

```
public abstract void graphUpdate()
```

Notifies this reconciler that the action-constraint graph was updated. The scheduler associated with this reconciler calls this method whenever a new fragment is added to the action-constraint graph that it maintains.

---

voteFor

```
public abstract void voteFor(ScheduleImpl schedule)
  throws java.io.IOException
```

Requests this reconciler to favor the specified schedule when making scheduling decisions. This method is the actual implementation of the `Document.voteFor(Schedule)` method.

---

receiveVote

```
public abstract void receiveVote(ScheduleImpl schedule)
```

Notifies this reconciler that a vote for the specified schedule has been received. The transmitter associated with this reconciler calls this method whenever it receveives a vote message.

fr.inria.gforge.telex.extensions
Class ScheduleGenerator

```
java.lang.Object
   |
   +-fr.inria.gforge.telex.extensions.ScheduleGenerator
```

Direct Known Subclasses:
IterableScheduleGenerator, ComponentIterableScheduleGenerator

---

public abstract class **ScheduleGenerator**
extends java.lang.Object

An object that generates sound `Schedule`s. Telex passes such an object to a `TelexApplication` through the `TelexApplication.execute(Document, ScheduleGenerator)` method when the specified document is updated, as described in `SchedulingParameters`. The application can then learn about the new state of the document by invoking the provided generator. When the application is finished, it must release the generator to indicate that it is ready to accept new generators.

In the general case, several sound schedules may exist for a given set of actions and constraints. A schedule generator should not generate just one schedule, but rather a set of alternative schedules. Indeed, the application must present these alternatives to user so that he can choose the one he prefers. A schedule generator should comply as much as possible with the following guidelines, by order of priority:

- only one of each set of equivalent schedules (according to *non-commuting* constraints) should be handed to the application,
- in case of conflict between actions of local user and that of remote users, schedules containing actions of local user should be handed to the application first,
- if the application has previously specified one or more *preferred* schedules, newly-generated schedules should be prefixed with one of the preferred schedules whenever possible,
- schedules should include as many of the actions of the action-constraint graph as possible.

A schedule generator is passed the action-constraint graph which it must compute sound schedules from. Most often, the actions of this graph belong to only one document. A schedule generator, however, must handle the case in which the graph contains the actions of several bound documents. In addition, an action-constraint graph may have several connected components. By definition of the ACF, the schedules generated on distinct components are independent. A schedule generator should let the application known that these schedules are independent. This allows the user to identify independent alternatives, and specify his choice for each of them.

Telex provides a default general-purpose generator: `DEFAULT_SCHEDULER`. A `TelexApplication` with specific needs may replace this default generator with its own by specifying it in its `ProcessingParameters`. The application-specific generator must extend the `ScheduleGenerator` class and provide a public nullary constructor.

Telex creates a new generator whenever it calls the execute() method. The generator is passed the following parameters: a snapshot of the action-constraint graph, a snasphot of the list of the documents this graph relates to and the generator that Telex created on the previous call to execute().

Author:
J-M. Busca INRIA/Regal

Field Summary

protected	`_activeFilters` The snapshot of the list of active filters to apply on the graph snapshot.
protected	`_graphSnapshot` The snapshot of the action-constraint graph to process.

protected	_previousGenerator The schedule generator that Telex previously created for the same graph.
protected	_relatedDocuments The snapshot of the list of documents the graph snapshot relates to.
public static final	DEFAULT_SCHEDULER The default schedule generator of Telex.

Constructor Summary

protected	ScheduleGenerator() Creates a schedule generator.

Method Summary

abstract ScheduleGenerator	getInstance(DocumentImpl document) Returns a schedule generator instance responsible for the specified document.
abstract boolean	isReleased() Determines whether this schedule generator has been released.
void	setParameters(TelexMultilog graph, DocumentImpl[] documents, ActionFilter[][] filters, ScheduleGenerator previous) Sets the parameters of this schedule generator.

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Fields

DEFAULT_SCHEDULER

`public static final java.lang.Class` **`DEFAULT_SCHEDULER`**

The default schedule generator of Telex. This constant is the `IterableScheduleGenerator` class.

_graphSnapshot

`protected fr.inria.gforge.telex.scheduling.TelexMultilog` **`_graphSnapshot`**

The snapshot of the action-constraint graph to process.

_relatedDocuments

`protected fr.inria.gforge.telex.implementation.DocumentImpl` **`_relatedDocuments`**

The snapshot of the list of documents the graph snapshot relates to.

_activeFilters

`protected fr.inria.gforge.telex.application.ActionFilter` **`_activeFilters`**

(continued from last page)

The snapshot of the list of active filters to apply on the graph snapshot.

_previousGenerator

```
protected fr.inria.gforge.telex.extensions.ScheduleGenerator _previousGenerator
```

The schedule generator that Telex previously created for the same graph.

Constructors

ScheduleGenerator

```
protected ScheduleGenerator()
```

Creates a schedule generator.

Methods

setParameters

```
public final void setParameters(TelexMultilog graph,
         DocumentImpl[] documents,
         ActionFilter[][] filters,
         ScheduleGenerator previous)
```

Sets the parameters of this schedule generator. Telex calls this method just after creating this schedule generator, and before any other method is called.

Parameters:

`graph` - the action-constraint graph to process.
`documents` - the set of documents this graph relates to.
`previous` - the schedule generator that Telex previously created on the graph.

getInstance

```
public abstract ScheduleGenerator getInstance(DocumentImpl document)
```

Returns a schedule generator instance responsible for the specified document. Telex calls this method for each document of the `_relatedDocuments` list, even if it contains only one element. Telex passes the returned schedule generator to the execute() method.

Parameters:

`document` - the document for which to return a schedule generator instance.

Returns:

a schedule generator instance.

isReleased

```
public abstract boolean isReleased()
```

Determines whether this schedule generator has been released. Telex calls this method to check whether it can call the `TelexApplication.execute(Document, ScheduleGenerator)` method again for the same document.

Returns:

true if this schedule has been released, and false otherwise.

fr.inria.gforge.telex.extensions
Class VotingReplicaReconciler

```
java.lang.Object
   |-fr.inria.gforge.telex.extensions.ReplicaReconciler
      |-fr.inria.gforge.telex.extensions.VotingReplicaReconciler
```

public class **VotingReplicaReconciler**
extends ReplicaReconciler

A distributed `ReplicaReconciler` based on voting. The reconciler works as follows. Each site proposes and votes for the schedule(s) of its choice. The schedule that receives a majority or a plurality of votes wins and is committed.

[To rephrase: sites actually exchange multilogs].
Author:
J-M. Busca INRIA/Regal

Fields inherited from class fr.inria.gforge.telex.extensions.ReplicaReconciler
_targetDocument, DEFAULT_RECONCILER

Constructor Summary

public	VotingReplicaReconciler()

Method Summary

void	graphUpdate()
void	receiveVote(ScheduleImpl vote)
void	voteFor(ScheduleImpl schedule)

Methods inherited from class fr.inria.gforge.telex.extensions.ReplicaReconciler
getScheduler, getTransmitter, graphUpdate, receiveVote, setParameters, voteFor

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructors

VotingReplicaReconciler

```
public VotingReplicaReconciler()
```

Methods

graphUpdate

```
public void graphUpdate()
```

Notifies this reconciler that the action-constraint graph was updated. The scheduler associated with this reconciler calls this method whenever a new fragment is added to the action-constraint graph that it maintains.

---

voteFor

```
public void voteFor(ScheduleImpl schedule)
  throws java.io.IOException
```

Requests this reconciler to favor the specified schedule when making scheduling decisions. This method is the actual implementation of the `Document.voteFor(Schedule)` method.

---

receiveVote

```
public void receiveVote(ScheduleImpl vote)
```

Notifies this reconciler that a vote for the specified schedule has been received. The transmitter associated with this reconciler calls this method whenever it receveives a vote message.

Index

V

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.