Access control in distributed file systems: design and realization on Pepys fs

Summary

This document describes the Access Control Model realized for the novel Pepys distributed, Internet-wide, file-system. The model design has been widely inspired to various existing standards and best practices about access control and security in file-system access, but it also echoes peculiar basic principles characterizing the design of Pepys, as well as the ΠP protocol, over which Pepys itself relies.

The proposed model is based of few cardinal points, which makes it particularly suitable for file-systems in which a consistently number of users wants (or has to) share its own data.

Particularly our model aims to create an environment where the following goals are satisfied:

- users can set their own rules for their data,
- the system administrator has to be able to set upper-bound rules in which every user has to obey,
- users can form real communities, sharing so their data and the rules applied to it,
- users can delegate other users in order to perform a given action on their behalf.

This document also provides technical details about how the model has been realized on a Linux port of Pepys.

Contents

Part I

Introduction

In computing, a distributed file-system (or network file-system) is any file-system that allows access to its objects (files and directories) from multiple hosts sharing via a computer network. This makes it possible for multiple users on multiple machines to share files and storage resources.

In a conventional file-system, is understood where the file actually resides; since the system and disk are known. In a distributed file-system, the location of a file, somewhere in the network, is hidden from the user point of view. Files in this type of file-systems are stored in *remote file-servers*.

Distributed file-systems may include facilities for file replication and fault tolerance. That is, if a limited number of nodes in the file-system crash, the system continues to work properly. File replication means that a file (or a file-system object in general) is replicated in more different servers; this fact affects positively performances and files availability.

Moreover a distributed file-system may provide a system file-local-caching. That is, when a remote file is retrieved by a user, such file is also stored in the user's machine for future accesses. This fact reduces the network traffic, by retaining recently accessed disk blocks in such cache, so that repeated accesses to the same information can be handled locally. If required data is not cached yet, a copy of data is brought from the server to the user.

Some example of well-known distributed file-systems are the *Jade file-system* [21] and the *sun network file-system* [25].

Since distributed file-system allows to share objects among different users, each of

them could (theoretically) have access to every object contained in it; a privacy issue is thus emerged. Therefore a protocol which regulates the users interactions and the object accesses has to be defined.

Usually each user owns a certain set of permissions on a given file-system object (capabilities) . A decision process uses these rules to establish whether a user can perform a requested action on a requested object. The protocol which rules how users can have access to the file-system objects is called *Access Control Model* (ACM).

Historically the first remote file-servers were developed in the 1970s. In 1976 Digital Equipment Corporation created the File Access Listener (FAL [5]), an implementation of the Data Access Protocol as part of DECnet Phase II which became the first widely used network file system. In 1985 Sun Microsystems created the file system called "Network File System" (NFS [27][24]) which became the first widely used Internet Protocol based network file system. Other notable network file systems are: Andrew File System (AFS [1]), Apple Filing Protocol (AFP [12]), NetWare Core Protocol (NCP [20]), and Server Message Block (SMB [16][17]) which is also known as Common Internet File System (CIFS [30][17]).

The presented work has been developed in Bell-Labs Ireland (Dublin) in July-November 2012. It was also submitted and then successfully presented at the 7th International workshop on plan 9 [9] the 16th in November 2012.

Part II

Related work

In order to design an efficient and state-of-the-art access control model, some of the most widely known and deployed standards for file-system access control have been considered.

Particularly, our work was greatly inspired to the POSIX Access Control Lists (ACLs) [2, 11]. POSIX ACLs overcome some of the limitations of the old UNIX file-system [28], allowing for the definition of multiple per-user and per-group rules, providing a great liberty of flexibility in expressing access-control rules. Even though the ACL model presents some limitations (as highlighted in [23]), they are used in the most modern operating systems. Some systems implement an abbreviated form of ACL by restricting the assignment of authorizations to a limited number of named group of users, like expressed by [23].

The access-control model proposed in this document is also based on attaching lists of access-control rules to files, therefore our model is also referred to as an ACL model, even though there are various differences with the standard POSIX ACL (see Section 2 for details).

In order to represent the set of allowed permissions for users or user groups, the classical concept of a *bit-mask* has been used, similarly to the UNIX file-system [28]. However, the set of allowed permission bits does not match perfectly UNIX. For example, we do not support the right of execution for files (that would not have sense in a distributed system); also, taking inspiration from NTFS [29], the *co-owner* bit has been added, used in ACL entries to define which users are co-owners of the file, i.e., they can manage its ACL settings.

Also, in our model the concepts of users and groups are somewhat unified, being also possible to define arbitrary nesting levels among groups of users. This behavior can be thought of as a flexible way to define users' roles and their hierarchical or nesting relationships, hence can be compared to the expressiveness often found in RBAC [26] models.

Our model design allows users to manage their own files permissions, allowing for a completely discretionary access-control, as found in DAC [15] models. At the same time, it is provided the possibility, for a system administrator (or specific set of privileges users), to define "upper-bound" rules that cannot be overcome by regular users, stealing some of the characteristics of typical MAC [15] models, and taking inspiration from similar characteristic available in in NTFS. In this way we provided a good trade-off between two different kind of policies: Discretionary and Mandatory. In the literature works as [4][3] and [32] have been also proposed, in order to find this trade-off.

Our implementation did not address comprehensively *authentication*, yet. However, a basic authentication mechanism has been realized, taking inspiration from HTTP-Auth [10], used in the HTTP protocol, in which clients send their hashed password to authenticate to the server. The authentication mechanism also re-uses the "everything is a file" old paradigm of UNIX and further developed in the Plan9 OS [7]. Furthermore, we support a primitive mechanism for *delegation [14]* of authority through off-line delegation certificates resembling Amoeba *capability lists* [18, 6]. Also an on-line delegation is available; in such case, as we will can see, certificates are not necessary.

Various other access-control models for distributed file-systems have been proposed in the literature, such as the WebFS [31] work, including a mechanism allowing entities to delegate other entities in order to act on their behalf ([8] [19] and [33]) on

a set of defined file-system objects, or others.

Part III

Access control in Pepys

1 Introduction on Pepys

Pepys is an innovative distributed file-system born to meet the increasingly growing demand, from users, to always have their data available anywhere.

Pepys is composed of a multitude of servers that, together, present a collection of files organized in trees or volumes. It uses a hierarchy of caching file *servers* and a set of archival storage servers, brought together through a common set of protocols for data access and control. Moreover, in order to design a fault-tolerant system, files may be replicated among servers; doing so it is even possible to improve the speed of files fetching (the same approach is also used by [31]).

In Pepys, when a new file is created, it is not necessary that every directory present into the path is present. For example, the file named `/a/b/f` can exist in the file-system without requiring existence of `/a/b` and/or `/a`. The existing object having a name with the longest prefix matching the name of another object merely becomes the *guard* of said other object. For example, if `/a` and `/a/b/f` exist and `/a/b` not, then `/a` is the guard of `/a/b/f`. The guard relationship among objects ultimately regulates how exactly access control is performed, within the Pepys file-system, as it will be detailed in section 2.

Pepys is a versioned file-system, i.e., when a file is modified, a new version of the file is added to the system, that keeps storing all the previous versions. This way it is always possible to keep track of the files history. Versioning allows for an efficient caching of files.

Moreover, files in Pepys may have *attributes*. These are defined in the same name space as for the regular files. For example, if `owner` is a valid attribute for the file `/a/b/f`, then its complete name is `/a/b/f/owner`. To avoid confusion between files and attributes, a special character is used in the file operations when referring to attributes.

Pepys is composed by two different types of process: ΠPclient and ΠPserver (see 1). The first one, as the name suggests, is a client-side process which represents the connected user and its requests of operations onto the file-system. The second one instead is a server-side process, which purpose is to process the user's requests, and to reply to him.

Pepys uses a new transport protocol (called ΠP) in order to minimize the round-trip message exchanges, between a ΠPclient and a ΠPserver, necessary to perform file operations. The protocol allows to send, to the server, multiple consecutive requests in a single packet (these request are packed by ΠPclients). Thanks to this approach, ΠPservers perform clients requests one by one, and send back a set of responses in order to communicate the operations outcome. In order to better understand how ΠP works, let us make an example. Let us suppose that a client wants to open a file named `test` and write into it a line of text. The used protocol allows to pack a single request which contains three statements: `open test; write line; close test`. After that the ΠPclient will send such packet to the ΠPserver, which will perform the required operations and, finally, it will send to the ΠPclient a response.

Being still under heavy development, Pepys has various features still under implementation, or merely at a design stage. For example, Pepys was not including any mechanism for access control, yet. This document describes the work that has been done in order to add an Access-Control Model to the Pepys distributed file-system, complying with the general principles behind the Pepys design.

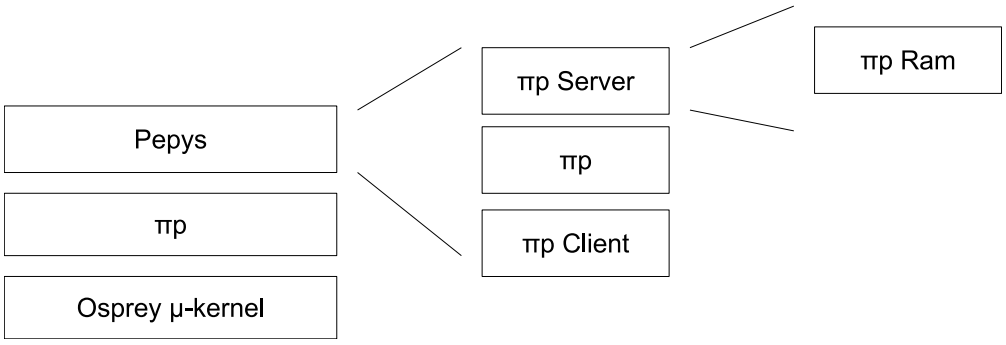

Figure 1: Pepys components.

Pepys file-system is currently implemented on top of a new operating system called *Osprey* [22] (see figure 1), providing an alternative approach to cloud computing, and specifically aiming to improve latency and predictability of cloud applications and support for mobility. In order to read a detailed explanation of Osprey, please refer to [22].

A key component in the overall architecture is the Π*P* protocol, supporting all Pepys operations, including various interactions with the Osprey kernel itself.

As showed in figure 1, the original Pepys server we modified included Π*P* Ram, basically an in-RAM file-system. Π*P* Ram contains data structures and functions to manage file-system objects and the accesses to them. Particularly it contains structures which represent files (or guards) and the necessary functions to create/write/read them. Such file-system is referred as non-persistent file-system, basically because its content cannot survive to a system reboot.

As explained in 3.1, this structure has been extended to keep files on a Linux (and generally POSIX) file-system, and to support our new AC model.

Finally, from now on, the terms *server* and *system* will be interchangeably used to refer the Π*Pserver*.

2 Access Control

In this section an overview of the access control model, with its main aspects, is shown.

Particularly, our access control model has to cover these aspects:

- each user is free to define the access control rules for its own objects, in the most flexible way possible;
- the traditional distinction between users and groups is replaced by a unified vision of such entities;
- access control rules can be specified at a generic abstraction level, considering sets of files and sets of users, then refined for specific subsets of those files and/or users;
- however, each user freedom is constrained by the rules dictated by system administrators, if any;
- re-using the "everything is a file" approach to manage as many operations as possible, including operations involving the administration of the access-control operations, such as editing of ACL rules or creation of users.

More details on the specific aspects are reported below.

2.1 Entities

The difference between users and groups has been overcome by introducing the concept of *entities*, representing users or groups of users, that can be authorized or denied the access to portions of the file-system.

An entity represents the subject who has request to perform an action onto the file-system. A trivial example could be this: let us suppose that user `Sam` wants to create a file; in such case `Sam`, as being the subject of the operation, is the entity. Every operation within the file-system is performed by entities.

In order to make the system security administration as scalable as possible, entities (i.e., users and groups) can belong to others entities. If needed, a system can be configured in such a way that a nesting relationship becomes valid when both involved entities agree about it. An entity has to be aware of the fact that, adding another entity in the set of entities belonging to it, is equivalent to giving them all the access rights to which it is entitled, unless otherwise overridden by more specific rules.

This feature can be also easily retrieved in the real life. Let us suppose that a person is registered at a rental movies center, and for this reason it owns a card that it has to show every time it wants to rent a movie. We can state that the set of people which own such card are the group of person which are able to rent a movie in the same movie center. Let us suppose now that the movie center belongs to a company which holds a set of equal centers. In this case a person will be able to rent movie in one of them without distinction. Of course exceptions are possible: for example the company can own different types of movie centers where different cards are requested. In this case, in order to rent movie in different movie centers, a person has to own different cards, hence it belongs to different groups.

In our approach there are no limitations for the nesting level of the belong-to relationship, which is to be considered a transitive relationship. For example if `Sam` belongs to `Nilo`, and `Nilo` belongs to `Tommaso`, then `Sam` belongs (indirectly) to `Tommaso`.

Hence, a "belong-to" relationship between two entities can be:

1. Direct

2. Indirect (if transitively inherited).

The first kind of relationship is considered stronger than the second one, from an access-control (AC) perspective, meaning that an AC rule referring to a direct father of a user has priority over an AC rule referring to a generic ancestor. The direct and indirect ancestors of an entity can be visualized in a "belong-to" relationship priority tree in which the entity under consideration is the root of the tree (see figure 2).

These trees (named belong-to trees or entities tree) represent the whole ancestor hierarchy of an entity. Trees are divided into levels, according the "belong-to" relationship priority, these level are scanned one by one during the access control algorithm 2.2.1.

Moreover, as we will see in section 2.3, an entity authentication is not mandatory. An entity can decide whether or not to authenticate itself onto the system. Of course the system will treat in different ways authenticated entities from those who are not.

Two system-level entities are always defined in the system, called `others` and `nobody`. Each entity defined in the system belongs implicitly to `nobody`, but only in the weakest possible sense (see Section 2.2.1). The `others` entity instead is a convenient way, in ACL rules, to refer to any authenticated user in the system. Also, unauthenticated entities, as well as entities just logged onto the system, and about to authenticate, are treated by the system as implicitly being the `nobody` entity. Also, is the system implicitly considers that `others` belongs to `nobody`, as shown in figure 2. Even though the name `nobody` could be misleading according what has just been said, is due to the behavior held by the system to the entities that are not authenticated (yet). Anyway the purpose of `nobody` and `others` is further detailed in section 3.6

Moreover since nesting of relationships can be arbitrarily added by users, loops

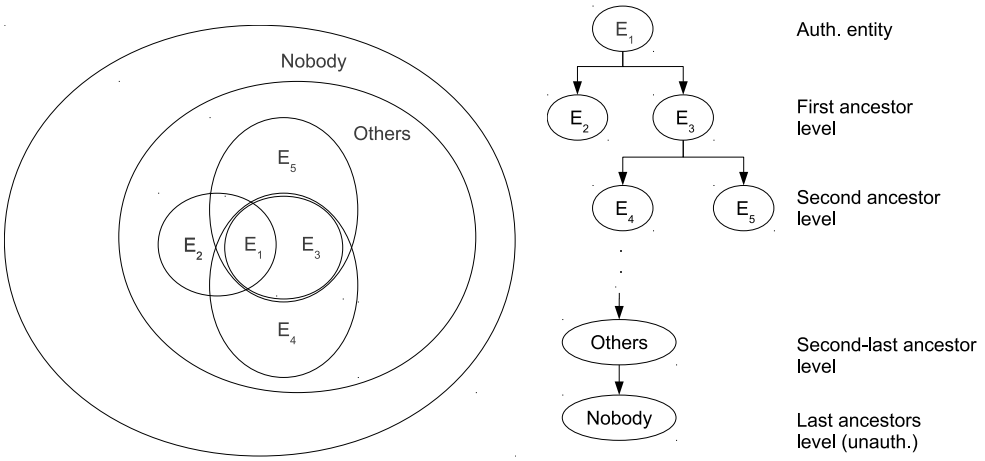

Figure 2: Belong-to relationship tree, rooted at a generic entity E_1.

are possible in the belong-to trees. Such a situation, albeit unusual, is still handled by the implementation consistently.

2.2 Access Control Model

Each object in the file-system owns an ACL table which contains the access rules governing access to it; each rule names an entity and its permissions to the object, as shown in figure 3.

Each ACL can have one or more *co-owners*, which can manage the rules in such ACL. At least one co-owner has to be always present, so to ensure that there is always someone able to manage the object security settings. Therefore, the system forbids the operation of deleting the ACL rule for the last co-owner. In other words if only one co-owner is named in a certain ACL, its entry cannot be removed until it is the only one contained in such ACL.

ACL rules apply generally to the object they are attached to, but are implicitly and dynamically inherited also by all the objects having it as a guard (i.e., the children files), and any other further object down the containment/guard hierarchy of objects

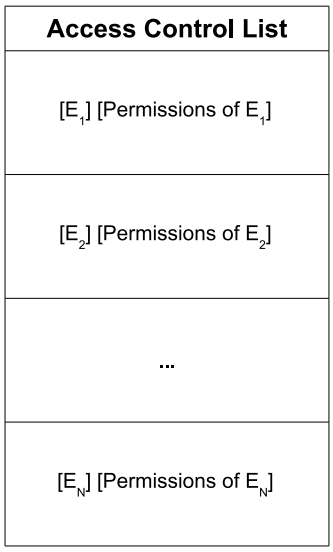

Figure 3: Access control list (ACL).

(i.e., the whole subtree rooted at the object).

Normally, a rule attached directly to an object takes precedence over a rule attached to its guard (father), or a rule attached to its guard's guard, etc. However, there is a special type of rules, called non-overridable rules (*o-rules*), that forcibly apply to the the whole subtree of the guarded files and cannot be overcome. Such rules are designed to be used typically by system administrators to restrict the AC settings that regular users may be willing to configure for their own created contents. As a result, a regular rule in an object stating that an entity has certain permissions is effective only if there are not any o-rules, in its guards chain or in the object itself, stating otherwise.

An ACL rule mentioning the `others` entity can be used to grant or deny access to any user known to the system, when acting as an authenticated user. Also, an ACL rule mentioning the `nobody` entity can be used to grant access to any user connected to the system, and if combined with `others` (see 2.2.1 and 2.3), for those who are not authenticated (yet). However, authentication is only partially addressed in Pepys

(e.g., server authentication is unaddressed, so far), as a full mechanism will have to be integrated with cryptography at the ΠP protocol level.

The type of supported permissions in the current design and implementation is mostly inspired to traditional UNIX file-systems: read and write of files, traversability of guards, ACL management (co-ownership) and delete permission (as found on NTFS file-systems [29]). However, this tentative set of permissions can easily be extended to more complex permissions or permission set. It is noteworthy to mention that, whilst on traditional file-systems, the read permission over a folder refers to the ability to read the folder contents, in Pepys it is planned to provide distinct permissions to read a guard's children (the guarded/contained objects), and to read any files contained in the corresponding sub-tree.

Another feature that is being discussed, from the ACM perspective, is the one in which there are multiple guards for the same object, a situation resembling the concept of link in traditional UNIX file-systems.

2.2.1 Decision Algorithm

At the core of the Pepys ACM there is the algorithm deciding whether or not to grant a given user access to a given file for a given operation. The central idea for such algorithm is: "more specific rules take precedence over more generic ones". This means that, if the entity can reach an object (permission given by the traversability bit), AC rules directly attached to it have priority over AC rules inherited by guard objects or other ancestors (the o-rules described above are the only exception, when present).

The decision algorithm locating the proper permissions applying to a given entity for a given operation (e.g., write) on a given object, can be expressed shortly in these few steps:

1. Traversability check: the system checks that the entity has the right to traverse (e.g., 'x' permission bit) all the existing guards going from the file-system root down to the desired object, looking at those guards ACL tables. The traversability permission, in such tables, can either be granted directly to the entity attempting the access, or indirectly through any of the entity parents or ancestors, in the belong-to relationship;

2. Check if there is a rule for the entity in the object ACL;

   (a) as soon as a match is found, its permissions mask are used to determine the access;

3. Check if there is a rule for any ancestor of the entity (i.e., as due to the belong-to specified relationships), giving priority to rules naming direct ancestors, then 2nd level ancestors, etc., up to `nobody`;

   (a) as soon as a match is found, its permissions mask are used to determines the access;

4. Get the inherited rules from the object guard and start again the algorithm from step 2;

5. If there are no rules about the entity or for one of its ancestors the access is denied.

It is important to say that the o-rules affecting a given entity are combined with the permissions mask returned by the algorithm above; this operation gives us the effective permissions which the entity owns on the object (see section 2.2.2).

Moreover, as we can see, it is been decided to give the priority to the entity ancestors, named in the specific object, rather than a possible rule for the applicant

entity in the object guard; this because we consider more accurate the rules contained in the specific object rather than those in its guard.

Furthermore, since every entity belong to `nobody` entity, if the `nobody` ACL rule is present, it allows to inhibit inheritance of guards rules; since the algorithm would break at step 3. Same reasoning can be made for `others` applies to every authenticated entity.

This makes the algorithm very flexible since is possible decide when the decisional process has to stop.

2.2.2 Access control rules

As reported briefly in section 2.2, there are two kind of rules.

- Regular rules (r-rules)
- Not overriddable rules (or obligatory rules or o-rules)

The first ones are the rules which are taken in account when we have to retrieve the most specific rule, for a named entity, during the decision algorithm. Instead the second ones can be considered merely as *upper-bound* rules which cannot be overcome by regular rules. As detailed in section 2.2.1, when an entity tries to perform an operation, the permissions as coming from regular rule matching its name are intersected with the restrictions provided by a matching o-rule.

Particularly, when an o-rule is defined in an ACL, it does not mean that the named entity in the rule owns the expressed permissions, but merely that such entity will not be granted more than those permissions, starting from the object where the o-rule is placed. Hence if the object is a guard, the entity will not have more than such permissions in the whole subtree held by the guard.

Furthermore, while using regular rules it is possible to add exceptions for a named entity in every level of a guards tree; o-rules can only narrow the permissions going down to such tree. For example, assuming that the path `a/b/c/.../f` exists, if the ACL of `b` contains an o-rule for a named entity, the rights for such entity (or its members) cannot increase going down the tree (hence `c/.../f`).

If an administrator wants to add an exception to an o-rule, it has to add it in the same ACL where the first o-rule is defined. For a detailed explanation see section 4.2.3.

The o-rules meaning is merely to contain the entity rights, inside a set of permissions. This makes the life easier for a system administrator.

O-rules have always priority on regular rules. Therefore if an ACL contains an o-rule which refers indirectly an entity and a r-rule which names directly such entity, both are taken in account, not solely the r-rule. The operation requested in such case is only allowed if the combination of both rules allows it.

Finally each entity can be directly named, in a rule (o-rule or r-rule), only once in an ACL.

2.2.3 Delegation

Entities can delegate others entities in order to act in their behalf, on a given object.

Each delegation is associated with a specific object and contains: the name of the delegator, the name of the delegatee, a set of permissions assigned to the delegatee, an expiration date and a value which represents the depth of the *trust chain* (see below in this section). Clearly, the permissions granted by delegation cannot be higher than the ones held by the delegator on the object. Figure 4 shows a general delegation and the fields contained in it.

Two kinds of delegation are possible:

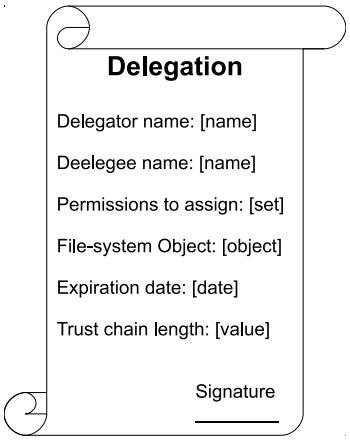

Figure 4: Delegation components.

1. On-line.

2. Off-line.

In the first one the delegator issues the delegation to the system, merely specifying in it who is the delegatee, its permissions, the depth of the trust chain and the file-system object on which the delegation is applied. In the second way the delegator, as well as fill the delegation like the first method, signs it and issues it to the delegatee. When the delegatee wants to perform an action on behalf of the delegator, it will present the delegation to the system. Figure 5 shows what has just been said: as we can see, on-line delegations are performed by a single step, whereas the off-line ones are performed by two differents steps.

In the first case the signature is not required, since the system knows exactly who is the delegator and its permissions (for the anonymous cases see 3.6). In the second case the system, before to approve the delegation, has to know who is the issuer. Therefore the delegation has to be signed by the delegator.

The delegations are taken in account using the same algorithm described in 2.2.1, and only if the access is denied using the regular ACL rules.

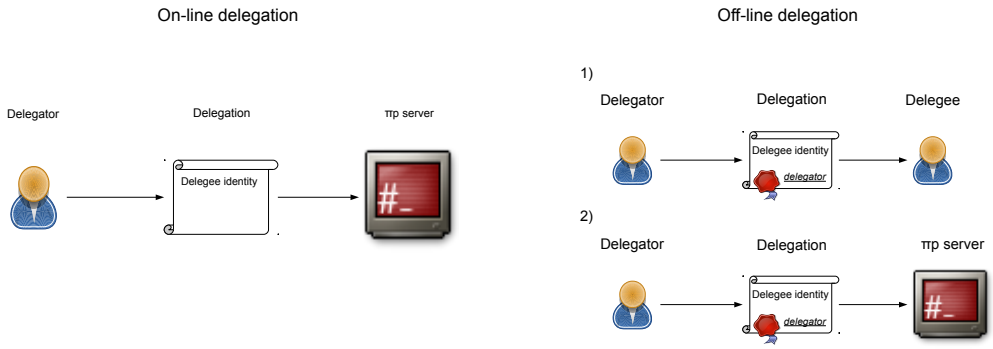

Figure 5: Types of delegation.

Every entity can use its delegations in order to delegate other entities, so as to build a chain of delegations (or trust chain). Every delegation in such chain cannot have an expiration time beyond the one expressed in the delegation immediately above (i.e., the delegation used to issue the new one). Therefore, the expiration time in the head of the chain bounds the expiration time of all the delegations in such chain.

Moreover, the entity which issues a delegation, marks it with a number representing the *maximum length* of the trust chain from that point. Of course these numbers, going down through the delegations chain, can only be smaller. Therefore the value assigned by the first delegator in the chain, represents the maximum length which the chain can have. This value can also be null, in this case the trust chain length is limitless. In figure 6 an example of trust chain in shown. As we can see the first delegator set a value as maximum trust chain length, which is decremented every time a sub-delegation is issued. Once this value reached zero, it will not be more possible to sub-delegate any entity.

Moreover it is always possible to revoke a delegation merely by resetting its expiration time; the delegation will become thus no more valid.

The type of explained delegation so far is referred as *grant delegation* in the lit-

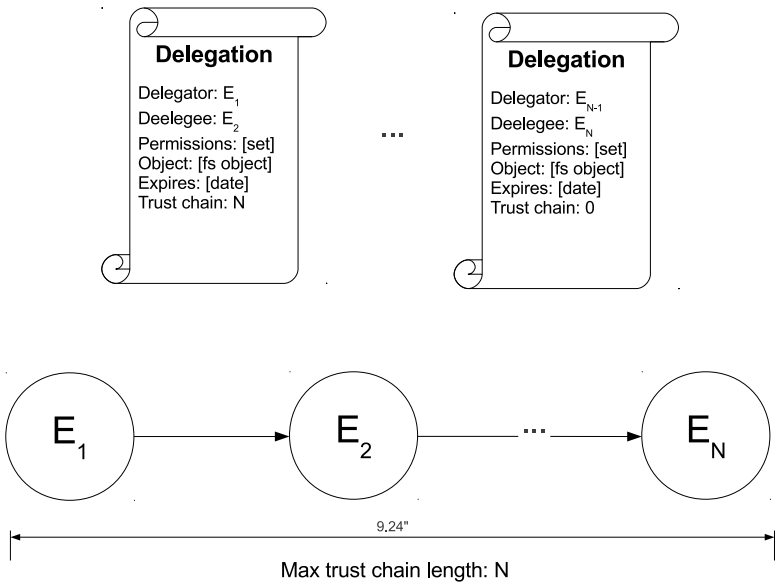

Figure 6: Trust-chain length representation.

erature; this kind of delegation is also treated in [8]. In fact according to such work, delegation of privileges may be classified into (at least) two different family: *grant* and *transfer*. In the first case, following a successful delegation operation, allows the delegated access rights to be available to both the delegator and the delegatee. In the second case instead, once a delegator delegated a set of rights to a delegee, it lost automatically such set of rights: a delegator does not able to benefit of the delegated rights. Our case is clearly the grant one. Such model is also referred as a *monotonic* model since a delegated entity rights cannot be decreased, once a delegation is successfully performed.

2.3 Authentication

Authentication of users has been temporarily realized as a simple (hashed) password verification. Authentication is not mandatory, to connect to the server. An entity

can have access to the system without having authenticated itself. In this case, the system considers the connected entity as being the `nobody` entity, thus the access-control permission specified for such entity throughout the file-system apply. While being connected to the system, an entity can authenticate itself whenever needed, upgrading its session from the rights corresponding to the only `nobody` entity to the rights associated with its actual name. Therefore until the user is recognized as to be `nobody`, the corresponding permissions will be taken in account in order to delegate other entities. Even though such delegations are (in most of the cases) purposeless, they are anyway available.

One of the goals of the Pepys file-system is to become a content-distribution platform. Supporting an unauthenticated state of the session is useful, in such context, to realize a sort of "incognito" mode of access by which public contents can be distributed worldwide without requiring users to reveal their identities.

Hence is clear that the system must be able to treat in a different way the authenticated entities from the other ones; this is achieved by using the couple `others` and `nobody`. Indeed `others` refers to entities which are logged onto the system, instead `nobody` to every entity present in the system (including both authenticated and not).

In order to better understand how these two entities are used, let us consider an ACL table of a file-system object. An ACL entry referring to the `others` entity applies to "every user logged and authenticated onto the system but for which no other ACL entries have been found in the ACL table"; an ACL entry referring to the `nobody` entity, instead, applies to "every user logged onto the system, either authenticated or not". ACL entries for `others` have priority over the ones for `nobody`, i.e., the AC engine behaves as if the former entity were a subgroup of the latter one (see figure 2).

Finally, if a server needs to authenticate users before allowing access to its contents, this can always be done by specifying the permissions wanted for the authen-

ticated entities following the rules above (and using `others` if needed) and no access rights for the `nobody` entity.

In literature some works which show a model where users are treated like the presented one can be retrieved. An example can be [13], where a special ACL entry to refer to every entity regardless of their credentials is used. Therefore such entry, whose subject is named *Anonymous*, matches every user in the system. The meaning of such entry is exactly the same of the Nobody one.

3 Implementation Notes

In this part some implementation notes are shown; unfortunately for company confidentiality questions, it is impossible to publish the source code.

3.1 Porting on Linux

The first step in our work was to unplug the Pepys file-system from its original structure (showed in figure 1) and therefore build a layer, called *Lib Posix*, in order to make the Pepys file-system runnable on UNIX machines.

In fact, thanks to the porting, we were able to use debug software, in order to better implement the presented access control model. Even though Osprey has an own set of tools, some these are not completed yet and also still under heavy development.

Moreover, in order to allow operations of swapping/loading objects from/into RAM, a new component, called *Pipdiskfs*, has been added to the POSIX version of Pepys server.

Particularly, thanks to the `Pipdiskfs` layer is possible to swap (or to load), from the RAM, whole ACLs tables or parts of them (for example few entries of a table). Similarly is also possible to swap (or to load) entire entities or merely parts of them. Of course is also possible to swap/load regular files.

Furthermore this layer fills the existent gap between the Pepys and the UNIX file-system. For example, guards in Pepys have been realized by directories in Linux.

As has been said in 1, Pepys is composed by two different parts: server and client. In order to let communicate these two parts, the porting has been realized by using the FIFO queues; provided by all UNIX file-systems. In this way clients and servers store their ΠP messages into these queues in order to communicate.

These queues are very simple: each process (ΠPclient or ΠPserver both) instan-

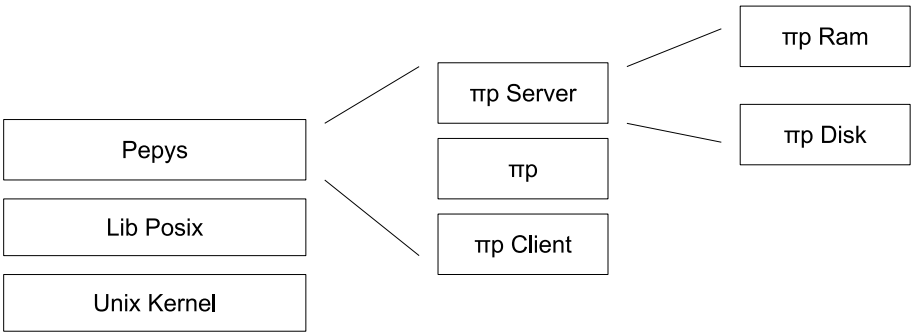

Figure 7: Porting on Linux implementation.

tiates an own *named pipe* (where the name of the process is the same of the pipe) which will be used to retrieve the incoming messages. In fact a named pipe (also referred as FIFO for its behavior) is system-persistent and exists beyond the life of the process. Processes generally attach to the named pipes (usually appearing as a file) to perform inter-process communication (IPC). In this kind of special-files each message is stored as if the file were a FIFO queue. When a Pepys client has to send a message to a server, it has to know the name of the server. Therefore the client will be able to perform a write operation on the server named pipe. Since different clients can communicate with the same server, they must be able to write in the the same named pipe without interfere each other. Therefore they must have exclusive access to the server named pipe each time the have to send a message. In order to let this possible a shared memory portion, in which processes can synchronize themselves, has been used. In order to see how a memory portion can be shared among processes, please refer to *shmget* manual.

The implementation is conceptually shown in figure 7.

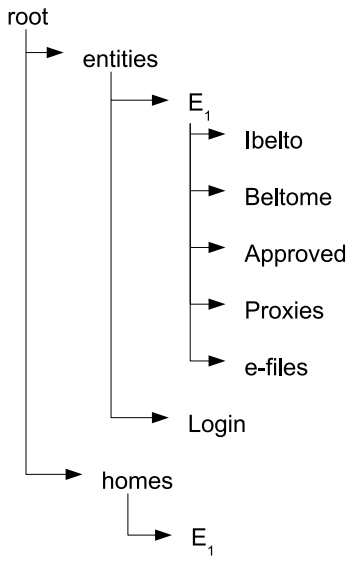

Figure 8: File-System structure.

3.2 File-System Structure

Inside the file-system, entities are represented by special guards, which own a set of special files. Moreover, as we can see in figure 8, each entity is associated with a `home` object (folder) over which it has full control.

Particularly each entity guard (e.g., E_1 in figure 8) holds:

Ibelto/Beltome: necessary to establish a new relationship.

Approved: list of entities which the named one belongs to

Proxies: provides a mechanism for permissions delegations.

e-files: others entity files as, for example, entity public key.

Each entity guard is managed by the guard above called `/entities`, which holds also a special file called `Login` in order to allow entities authentication.

An ACL table is represented by an object attribute, which can be changed only by the co-owners as reported in such ACL.

Instead an object delegation is a special object attribute, managed by the system, and hidden from the user's point of view.

The motivations behind this implementation is discussed in section 2.2.3.

3.3 Entities relationship

When an entity wants to become member of another entity's users group, it writes the name of the other entity in its `Ibelto` file (over which it normally has write permission). The other entity (or the system administrator), on its own, has to write the name of the first entity in its `Beltome` file, in order for the new relationship to become effective. For each entity, the effective `Ibelto` relationships are reported in the `approved` special object within the entity folder, normally accessible to it for reading.

It is impossible for an entity to remove from its parents the system entities `nobody` or `others`. Also, depending on how the system is being administered, it is possible to allow users to write to their own `Ibelto` file, enabling them to propose changes to their belong-to relationship, including their removal from groups they belong to. On the other hand, it is equally possible to forbid such write operations, leaving the administration of users and groups entirely to system administrators, as it commonly happens in nowadays operating systems.

In order to express the willingness to belong to another entity, a user has to write in its `ibelto` file the entity name, using this syntax:

```
[first entity name]:[second entity name]:...:[last entity name]
```

Consequently, in order to express the willingness to be the *father* of an entity, a user has to write its name in the `beltome` file, using the same syntax.

In our implementation, when the agreement is reached, the relationship become

effective.

3.4 Collection of entity rules

One of the biggest challenges was to dynamically collect the o-rules, referred to a given entity, without heavily impact on the model performances.

Since these kind of rules have the highest priority on every one else, they have to be taken in consideration every time they change. Even though a new o-rule could be propagated in all the objects held from the one where is applied, this would heavily impact in system performances, by making really slow any rules modification.

In order to find a good trade-off between adaptiveness and performances, o-rules are collected during the walk procedure; which is performed to check if an entity is able to reach the file-system object in which the operation has been requested. This seems to be a really good solution since, in any case, an entity has always to walk to reach an object. Hence the retrieved rules can be easily collected.

Anyway this approach implies a drawback. Let us suppose that the path `/a/b/c/d/.../f` exist, and an entity walks from `a` to the object `d`. During the walk the o-rules are collected. Let us assume that such entity, once it reached the object, uses it but leaves it opened (e.g, let us think about a file-system window in which a user browses files and directories). Even thought a new o-rule (about such entity) is added in the object `a`, next entity walk will start from the object `d` (it is left opened), hence the new o-rule will not be collected. In order to avoid these situations, every time a new o-rule is added, all the walk operations (of every entity) should start from the file-system root path. This solution would imply, clearly, a heavy performances leakage. Therefore the collected o-rules are refreshed when an entity visits an object. Moreover, when an entity closes the object reached by a walk, its next walk will start from the root

guard, hence o-rules will be collected again.

Of course an entity can have more than one opened object. In this case if it closes one, it is always able to continue to walk using another opened object (let us think again to have two different file-system windows).

Moreover the proposed solution does not impact on the model consistence. In fact it is lawful suppose that if an entity is able to perform an action on an object, it has the right permissions to do so. Therefore it should have such permissions for the time the object is left opened.

In order to define if an entity owns the walk permission, during the walking procedure, (also) a regular rule in every visited object has to be searched. Since we have to pick up a valid r-rule from the object ACL table, the rule we obtain is also stored, so to have it already available during the access control algorithm. This impacts positively on ACM performances in terms of execution time and complexity.

The caches are realized by a stack of rules. In this way when a user performs a forward walk (e.g., from `/a/b` to `/a/b/c`) a new rule is pushed at the top of the stack. When a user performs a backward walk (e.g., from `/a/b/c` to `/a/b`) the top stored rule is popped out. By using such approach the first top rule is the one concerning the very object which the user is requesting.

Moreover when applying the algorithm 2.2.1 is requested to search for a rule in the object guard, we do not need to visit such object in finding a rule. All we need to do is a pop operation on the stack-cache in order to retrieve a valid rule.

Cases in which no rules have been found in an ACL, for a user, in the walk operation, the last valid one is replicated.

Furthermore stack-caches are stored in the user session structure; and there is one for each opened file-system object. Clearly whether a user performs a login, these caches have to be completely invalidated, since its credentials are changed.

Another prefixed goal was to let the delegation mechanism as an addictive optional feature in our model. In order to do not impact on ACM performances, delegated rules are not stored during the walks procedure. Delegations are checked dynamically only if the requested action by an entity, according the ACL regular rules, would be denied.

3.5 Delegation

As we said in section 2.2.3, two kinds of delegation are possible: on-line and off-line. In both of these two cases the delegatee acts in behalf of the delegator.

A valid delegation acts like a temporary ACL entry, so the very first idea could be to change (temporary) the ACL tables to store a delegation rule. However, the delegator might not have permissions to administer an object ACL but still willing to delegate some other entity to perform actions on its behalf on that object.

The proposed model allows for this kind of scenarios, merely allowing to each entity to solely write/read its own `proxies` files. As a consequence, the system will make the requested delegations effective, or ignore them, if they are invalid.

Moreover a delegation can contain a subset of permissions held by the delegator on a given object. Therefore it is not mandatory to delegate an entity with all the permission held by the delegator. For example let us suppose that `entity1` wants to delegate `entity2` on the object `file` in which `entity1` owns the whole set of permissions (i.e., cdrwx). `entity1` can only delegate `entity2` to read, not necessarily to have the same rights of `entity1`.

On-line delegation In the on-line delegation, when an entity wants to delegate other entities to act in its behalf, it has to write the delegation in its `proxies` file. Specifically it has to indicate who is the delegatee, its permission, an expiration date,

depth of delegation chain and the object to which the delegator is referring to.

After that, the system will consider the delegation as effective only if it is compliant with the delegator's permissions on the specified object (i.e., an entity cannot delegate permissions it does not possess over a file-system object).

The right syntax to use to issue an on-line delegation is:

```
[delegatee name]:[permissions]:[depth chain]:[expiration date]:[object]
```

Let us suppose therefore that `entity1` wants to delegate `entity2` (and only it) to read an object named `file` up to 14th in November 2013. `proxies` file has to contain the following line:

```
entity2:r:0:00/00/00/14/11/2013:file
```

Off-line delegation In the off-line delegation method, the delegator composes a delegation, it signs it, and it stores it somewhere in the file-system. The delegatee must specify in its `proxies` file who is the delegator and a valid path where the signed delegation is stored. The system then checks the delegation signature using the public delegation key available in the entity folder, and, only if the verification succeeds, the delegation is considered effective.

Therefore, this time, the `proxies` file has to contain two fields, specifically:

```
[delegator name]:[path where the delegation is stored]
```

In the example mentioned in the above paragraph, by supposing that the delegator stores its delegation in the path `/deleg/to_entity2`, `entity2` has to write in the `proxies` file:

```
entity1:/deleg/to_entity2
```

The first information is necessary to the system in oder to retrieve the correct

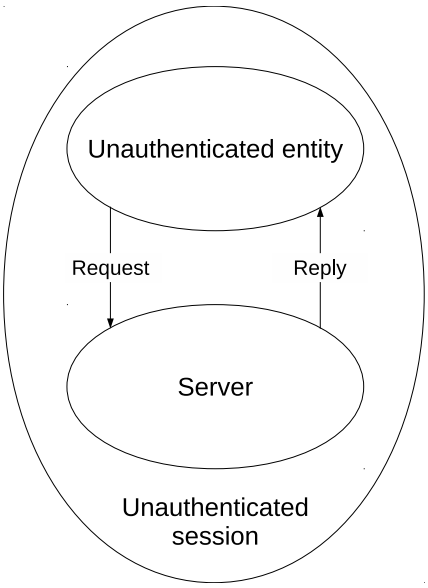

Figure 9: Unauthenticated session.

public key (in order to verify the delegation signature); the second one to know where the delegation can be found.

The syntax of the delegation is the same showed in the above paragraph:

```
[delegatee name]:[permissions]:[depth chain]:[expiration date]:[object]
```

3.6 Authentication

When a client logs onto a Pepys server, it is not required to authenticate itself immediately, resulting in a session being in an unauthenticated state (see figure 9).

This means that the `nobody` access rights apply for the client, whenever an operation on the file-system is attempted. The client can authenticate itself at any time by using the special file `Login`. Specifically, when an entity wants to upgrade its session (see figure 10), it has to write its (SHA-256) hashed password using a write command.

In section 6.8 will be showed how the login procedure has been implemented.

The server compares the hashed password with the one stored in the entity pass-

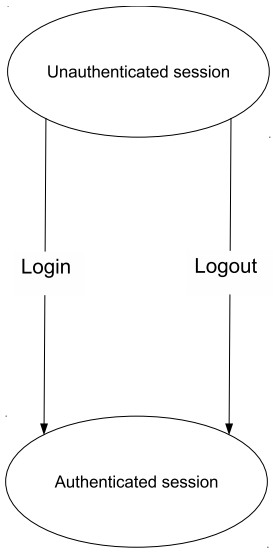

Figure 10: Session switching.

word file, and, if they match, the entity session is upgraded to an authenticated state. From now on, the actual identity of the entity is used for checking the access rights of the user.

Note that the `Login` file is a special file, in that it does not really stored any password. Such file can be opened by multiple remote clients concurrently without problems, as in the implementation the authentication material being provided by each client is kept into a separate buffer associated with each session.

Moreover thanks to the characteristics of the ΠP protocol to group multiple requests in the same message, it is possible for a remote client to stuff, within a single round-trip interaction with a Pepys server, the set-up of a session, opening of the `Login` file and writing of the password, opening of the target file-system file and issue of the desired read or write operation.

However, the very simple authentication protocol realized so far is also relatively weak, in that it is easily subject to replay attacks, thus it can be improved by adding a time-stamp to the hashed password to be written into the `Login` file, or a server-

provided random number (i.e., a *nonce*). Though, the last mechanism would require at least two round-trips with the server.

Finally, we plan to review and improve the authentication mechanism by integrating it with cryptographic extensions of the ΠP protocol which are being designed at the time of writing, that will allow for having encrypted client-server interactions.

3.7 File-system tools

The developed software included, in addition to the Pepys file-system porting, a few other tools:

1. Administration tool allowing for initializing the file-system, specifying:
   (a) Entities allowed and their login password.
   (b) Server name.
   (c) Mount point (on the underlying Linux file-system) to allow for swapping/loading of objects from/into the RAM.
   (d) Path of directory that will contain temporary files (i.e., named FIFOs currently used for client/server communications).
2. An interactive terminal in which is possible interact with the Pepys file-system (create files, administer ACL settings).
3. A set of "ad-hoc tests" to test the main file-system features.

In the next sections the purpose of each of these will be explained in detail.

3.7.1 Configuration file

In order to ease the configuration of the file-system, an administration tool has been created.

Particularly, a file in which specify the allowed users and some of initials configurations, has been provided.

Let us see an example:

```
1  # Example of config file
2  # This is a comment line
3
4  # Server name
5  server: sam
6
7  # Set the disk mount point
8  mount: ~/Alcatel-Lucent/Pepys/Porting/DISK/root
9
10 # Set the keys directory
11 # Public keys mast be named: pubkey
12 # Private keys (if there are) must be named: privkey
13 # This path must contain directories named as the represented entities
14 keys: ~/Alcatel-Lucent/Pepys/Porting/DISK/RSA_keys
15
16 # Entities allowed
17 # Format: entity: Name Password
18 entity: entity1 asd
19 entity: entity2 asd
20 entity: entity3 asd
21
22 # Temp file path
23 tmpdir: /tmp/
```

Server name is used to represent the administrator entity within the file-system, and in order to let the clients able to exchange ΠP messages with the server part.

The Pepys file-system mount point is expressed using the key-word `mount`. Inside

such path is possible see the Pepys file-system structure mapped on Linux, as said in 3.1.

The key-word `key` represents the path where the entities keys are stored. Specifically such path has to contain a set of different folders which are named as the entities they represent. For example, according the above example of configuration file, let us suppose that the system has to own the public keys of the entities: `entity1, entity2` and `entity3`. The path contained in `keys` has to contain three directories (or four if also the key of the serer has to be stored) named `entity1, entity2` and `entity3` (and `sam` if necessary). Each of these directories, in turn, contain the public keys of the entity (renamed `pubkey`), therefore: `$keys/entity1/pubkey`, `$keys/entity2/pubkey` and `$keys/entity3/pubkey`.

The allowed Entities in the file-system are specified by the key-word `entity`. Each entry has to contain the entity name and a password.

Finally is possible to specify the path where the FIFO queues are stored, by using the key-word `tmpdir`.

It is important to say that none of these key-words are mandatory; if one or more of these are not expressed, the default values are used.

Anyway if no entities are reported, no one will be able to authenticate itself. Every entity in the system will be referred as the `Nobody` one within the system.

3.7.2 Interactive terminal

In order to access to the file-system, an ad-hoc terminal has been implemented.

Specifically some of the well-known Linux command has been replicated: `echo`, `cat`, `> (»)` (output redirection), `seftacl` and `getfacl`. Moreover some new command are been added: `login`, `logout` and `help`.

Echo The `echo` command prints a line of text in the standard output.

The syntax is:

```
echo line-of-text
```

Specifically using the quotes is possible to write, in the terminal, the exact line of text contained between them.

Furthermore, by combining the `echo` command with the `> (»)`, is possible to write in a file. For example let us suppose that we want to write the phrase `Hello world!` in a file named `asd.txt`. In order to do so, the the right syntax to use is:

```
echo "Hello world!" > asd.txt
```

The execution of above line performs a trunc mode write, hence a new phrase is stored to the file, truncating its whole content after that.

In order to append a new content at the end of a file, the symbol `»`, combined with the command `echo`, has to be used. For example, let us suppose that we want to write `My name is Nilo` at the end of the same file. What we have to do is write into the terminal:

```
echo "My name is Nilo" >> asd.txt
```

Finally, using the `echo` command, if the indicated file does not exist, it is created.

Cat The `cat` command, shows on the standard output the content of a file, which is passed as argument.

The syntax is:

```
cat file
```

For example, if we use the cat command with the file `asd.txt` (used in the echo paragraph):

```
cat asd.txt

asd.txt:
Hello world!My name is Nilo
```

If the file does not exist, an error is returned.

Setfacl This command allows to add (and to delete) rules in ACL tables. `setfacl` has been implemented regarding the POSIX one.

The syntax is:

```
setfacl [-m/-x][entity][:[permissions]] object
```

The option `-m` is used to add (or to modify) an ACL rule. In this case is mandatory to express the entity name, the permissions to give to such entity and the file-system object where the rule applies.

The option `-x` is used to delete an ACL rule. In such case the solely mandatory values are the entity name and the file-system object.

Like already said in 2.2, the available permissions are: read (r), write (w), traversability of guards (x), ACL management (c) and delete (d).

Let us assume that we want to give the write permission to `tom` in the file `asd.txt`. The right command to use is:

```
setfacl -m tom:w asd.txt
```

Note that `setfacl` command used with `-m` does not always add a new rule in an ACL table. Indeed, like said in section 2.2.2, if a rule which names the same entity is already present; it is updated with the new given permissions.

In order to remove an ACL rule, `-x` option has to be used. For example if we want to remove the rules which names `tom` from the ACL table of `asd.txt`, we have to use the following syntax:

```
setfacl -x tom asd.txt
```

Note that whether no rules which name the given entity are present, no errors are returned.

Overall using `setfacl`, if the object does not exist, an error is returned.

Getfacl The command `getfacl` is used to display the current ACL table of a file-system object.

The syntax is:

```
getfacl object
```

For example, let us suppose that we want to see the ACL table of `asd.txt` used in the last paragraph, the right syntax to use is:

```
getfacl asd.txt
```

Assuming that `asd.txt` was created by the entity `nilo` and that we did not remove `tom` among the ACL rules in the last paragraph, the result that `getfacl` shows will be:

```
**** ACL TABLE ****
-crw- nilo
---w- tom
**** END TABLE ****
```

According this ACL table, `nilo` is the only file co-owner and it can (of course) change the ACL rules, read and write the file. Instead the only action which `tom` is able to perform is to write into the file.

Finally note that no one of the named entities owns the `x` permission. This because traversability cannot be applied to files, and the execution mode is not contemplated according what has been said in II.

Login The `login` command allows entities to authenticate themselves to the system.

Therefore an effect of th`login` command is to upgrade an entity session, from an authenticated one to the authenticated one, like it has been explained in section 3.6.

The syntax is:

```
login password
```

The purpose of this command is merely to be a shortcut. Indeed, as reported in 3.6, the authentication process is performed by writing a hashed password in the `login` special file. Therefore this operation can be accomplished even by using the `echo` command, if the hash of the password is known.

The steps performed by the `login` command are: hashing the passed password as argument and to write the result into the `login` special file.

Note that, like expressed in 3.6, no real file write operation is performed. `login` special file is merely an interface to let the entities able to authenticate themselves.

Logout The `logout` command allows entities to de-authenticate themselves from the system.

The syntax is:

```
logout
```

Thanks to this command a session of an entity is downgraded to the unauthenticated one. Therefore from now on the entity will be recognized as `nobody` (as explained in section 3.6), until it authenticates it again.

Help Finally is also possible to see all this commands, with a little manual, by using the `help` command.

The syntax is:

`help [(opt)command]`

This command, without arguments show all the available commands with a short description; otherwise it shows the command passed as argument with its description.

3.8 Ad-hoc tests

In order to test the behavior of the access control model, a set of ad-hoc tests are provided.

These tests merely use the commands provided by the Pepys terminal, in order to reproduce real situations, and to test the results given by the access control model.

The provided tests aim to cover every aspect of the proposed model, let us see now each of them in detail.

Every test returns an OK message if all went well, FAIL if (at least) an error occurred.

Note that each entity, generally, is only able to write starting from its own home directory (see figure 8). In order to do not repeat the part of such path every time we have to perform an action on a file, the first part will be omitted. For example, in order to have access to a file placed in `/homes/nilo/a/b/file`, we will use the compressed path form `/a/b/file`.

Permission denied In this test there are two involved entities. The first one, named `entity1` creates a file, then changes the access control rules in order to deny the read access to another one entity named `entity2`.

After that, the entity `entity2`, logs onto the system and tries to read the same file.

The test succeeded if such operation is denied.

Entity nesting Entity nesting allows the permissions inheritance among entities, as explained in section 2.1. In order to test this behavior an ad-hoc test was created.

In this test three entities are involved: `entity1`, `entity2` and `entity3`.

First step is to create a relationship between `entity1` and `entity2`; particularly `entity2` becomes a child of `entity1`. Same operation is performed between `entity2` and `entity3`. Therefore the final nested relationship `entity3` $\in$ `entity2` $\in$ `entity1` has been created.

After that `entity1` creates a new file in which it writes a line of text. After that `entity3` logs onto the system and tries to write in the same file.

The test succeeded if such operation is allowed.

Rules overriding In this test the o-rule behavior is shown. Three entities are involved: `entity1`, `entity2` and `entity3`.

First of all an entity named `entity1` logs onto the system and creates a file `/a/b/file`.

Note that for what has been said in section 1, it does not necessary that `a` and/or `b` exist; the last existent one become automatically the guard of the new created object. Anyway, since in this test also the object `b` is created, it becomes automatically the `file` guard.

Moreover `entity1` add an o-rule to the object `b` where denies to an entity named `entity3` to gain the read access from now on (i.e., beyond `b`.

Furthermore `entity1` adds a regular rule in the ACL of `b` which states that the named entity `entity2` will be able to manage such ACL table.

After that `entity2` logs onto the system and set an new rule in the `file` ACL (it is able to do it according to the algorithm showed in section 2.2.1) in which gives the read permission to `entity3`.

After that `entity3` logs onto the system and tries to read the content of `file`.

According the rule's precedence, every entity which belongs to `entity3` (including the entity itself), must not be able to read any object beyond `b`.

The test succeeded if the read operation requested by `entity3` is denied.

Priority relationship This test case is thought to show the correct behavior of the algorithm 2.2.1.

The first step is to create the relationship: `entity3` $\in$ `entity2` $\in$ `entity1` (as it has been shown in the entity nesting paragraph).

After that an entity named `entity4` creates a new file in which set two different rules, like reported below:

```
## ACL table ##
entity4: cdrw-
entity1: -----
entity2: --r--
```

After that `entity3` logs onto the system and attempts to read the file. By scanning sequentially the ACL table, the first match for `entity3` would be the one represented by the second line. Even though this would be a valid match, according the `entity3` entities-tree, the third line in the ACL table represents a more specific rule for `entity3`.

Therefore, in order to consider the test passed successfully, the decision algorithm 2.2.1 has to pick the third rule from the ACL table. Hence the test succeeded if `entity3` is able to read the file.

This test merely proves the priority rightness of the algorithm showed in section 2.2.1.

On-line delegation In this test the on-line delegation behavior is tested.

Three entities are involved: `entity1`, `entity2` and `entity3`.

`entity1` creates a new file in which denies the read operations to `entity3`, but gives such right to `entity2`.

After that, `entity2` logs onto the system and issues an on-line delegation to `entity3`, where `entity2` gives to `entity3` the permission to read in behalf of `entity2`; for two seconds.

Therefore `entity3` logs onto the system and tries to read the content of the file.

If the operation succeeded, `entity3` waits for two seconds, then retries the same operation.

If the first read succeeded and the second one does not, the test succeeded.

Off-line delegation This test is very similar to the one explained in the above paragraph.

In this case `entity2` signs an off-line delegation and stores it in the path `/a/b/delegation`.

After that the same test showed in the last paragraph is repeated, using this kind of delegation.

Again, if the first attempt of read performed by `entity3` succeeded and the second one fail, the test succeeded.

4 Some examples

In the proposed ACM, a good trade-off between flexibility and simplicity of setup has been targeted. As it has been said, this goal was achieved by merging the concepts of users and groups in the solely concept of entities and by adding the concept of o-rules.

Let us see now some examples which show how this model aims to face some common situations. In order to better understand when we are talking about a group (i.e., an entity which contains at least another entity) the expression *g-entity* will be used.

4.1 Deny access to a defined set of users

It is common that a system administrator wants to deny the access to a particular directory tree (guards in Pepys), or merely to a given file, from a user or a group of them.

Let us suppose that the administrator does not want a `g-entity` to be able to write on a defined file, clearly supposing that `g-entity` (and each entity which belongs to it) is not a file co-owner.

Instead of adding a new rule for each entity belonging to `g-entity` (even though rightful), thanks to the entity concept itself, this situation can be solved merely adding a new rule in the file ACL.

Assuming that the administrator name is `Admin`, the ACL table is:

```
## ACL table ##
Admin:     cdrw
g-entity:  --r-
```

Of course it is also possible to add exceptions for specified entities which belong to the `g-entity`.

For example if the administrator wants that only `x-entity`, which belongs to `g-entity`, is able to write on such file, a new rule has to be added:

```
## ACL table ##
Admin:     cdrw-
g-entity:  --r--
x-entity:  --rw-
```

Hence when an entity which belongs to `g-entity`, but different to `x-entity`, tries to write in the file, the access will be denied. Instead for `x-entity` will be grant.

It is also important to remark that the order in which these two rules are placed in the ACL table is irrelevant for the algorithm outcome.

The case in which a file co-owner belongs to `g-entity` is meaningless in this scenario; because being a co-owner, it would be able to change the rule. We will see in 4.2 how these kind of problems can be addressed.

4.2 Not overridable rules

Another possible scenario may be one in which an administrator wants to let the users free to manage the access rules for their objects, but restricting the rights which they can have/assign.

Assuming the path `a/b/c/d/.../f` exists; let us see some examples which show how an administrator can manage the access rights to the objects contained in such path.

Moreover, the whole file-system subtree rooted by (for example) the guard `b`, will

be indicated using the expression "beyond the guard `b`".

4.2.1 Rights restriction

Let us suppose that `g-entity1` is a co-owners of guard `c`:

```
## ACL of c ##
Admin:      cdrwx
g-entity1:  cdrwx
```

Let us suppose, moreover, that the administrator wants to establish a rule for the entity `g-entity2`:

- `g-entity2` must not be able to delete objects starting from b.

In order to achieve this goal, the administrator, has to add an o-rule for `g-entity2` in the ACL table of b:

```
## ACL of b ##
Admin:       cdrwx
g-entity2: !c-rwx #o-rule
```

In this way, the delete right to every entities belonging to `g-entity2` starting from the guard `b`, is denied.

This does not mean that `g-entity1` cannot assign the delete permission to `g-entity2` (or at one of its members); but merely that if it does that, it will not take effect:

```
## ACL of d ##
Admin:      cdrwx
g-entity2:  cdrwx #d bit does not take effect
```

In the last table g-entity2 being a co-owner of the d, can assign the delete permission to other entities. Therefore such entities will be able to delete the objects where they own such permission. This does not create inconsistencies since, according to the o-rule expressed in b, g-entity2 can have the rights of co-owner and only it (and its members) has to be unable to delete objects.

4.2.2 Contained rights

Let us suppose now that the administrator wants to establish these rules that cannot be overcame beyond the guard b:

- g-entity1 and g-entity2 are the only co-owners allowed (together with their members).
- Deny the delete permission to anyone else.

In order to do so, the administrator has to add three rules in the b ACL:

```
## ACL of b ##
Admin:       cdrwx
g-entity1: !cdrwx #o-rule
g-entity2: !cdrwx #o-rule
nobody:    !--rwx #o-rule
```

The first two rules are necessary because, otherwise, the system would pick the nobody o-rules also when g-entity1 (or g-entity2) walks through the guard b. In fact as expressed in 2.1, every entity in the system belongs to nobody.

4.2.3 The o-rules exceptions

Using the o-rules is impossible to add exceptions for a named entity beyond the first one. This fact means that if an o-rule applies for an entity in the guard `b`, the permissions held by the entities which belong to it (and the entity itself) cannot be increased in the guards `c`, or `d` etc. Permissions expressed by these rules can only be decreased, over the first one, going down to the same file-system tree.

For example, let us suppose that `x-entity` belongs to `g-entity1`, and that the administrator wants to establish these rules beyond `b`:

- Deny to `g-entity1` to delete the objects.
- Let free `x-entity` to delete the objects.

In this case the table is:

```
## ACL of b ##
Admin:       cdrwx
g-entity1: !c-rwx #o-rule
x-entity:  !cdrwx #o-rule
```

Therefore each member of `g-entity1`, except for `x-entity`, will not be able to delete objects beyond `b`.

Again, if the o-rule for `x-entity` would be wrote in the ACL of the guard `c`, it would not be taken under consideration.

We will see in section 6.6 why is not possible add an exception to an o-rule by using a r-rule.

4.2.4 Freedom of decision

This is example is a merely generalization of those ones above. We show how an administrator can let the users free to decide how to manage their objects (so who are the co-owner etc), but prohibiting to anyone to accomplish a given action.

Let us assume that the administrator does not want that anyone deletes the objects beyond the guard `b`; but for the remain rights the users can decide for themselves.

In such case the ACL table should be:

```
## ACL of b ##
Admin:   cdrwx
nobody: !c-rwx #o-rule
```

Furthermore, the rule above does not implies that everybody in the system (represented by `nobody`) can manage the objects beyond `b`, but merely that the objects co-ownership is denied to no one.

If only the logged users can be declared as co-owners, a new rule rule has to be added:

```
## ACL of b ##
Admin:   cdrwx
others: !c-rwx #o-rule
nobody: !--rwx #o-rule
```

4.3 Sticky bit example

This example shows how to implement a behavior sticky bit like.

Our goal is to let the logged users free to create their own objects, in a given

directory, but to make they able to delete only the ones they are created. In our model the file-system object creation is associated with w bit.

Assuming that the path `/a/` exists, let us see how is possible achieve the prefixed goal.

The ACL table of `a` should be:

```
## ACL of a ##
Admin:    cdrwx
others:   --rwx #o-rule
nobody: !---- #o-rule
```

According to this table, each authenticated entity is able to create objects in `a`. Particularly when a new object is created (assuming by `x-entity`), its ACL, by default, is:

```
## ACL of a generic object ##
x-entity: cdrwx
```

With these two tables, entities different from the one who has created the file-system object, will not be able to delete such object.

In this case, anyway, every entity is able to write in every object contained in `a`. If we want that only the owner may be able to do so, it has to add a new rule in its objects.

```
## ACL of a generic object ##
x-entity: cdrwx
others:   --r-x
```

Therefore, unfortunately, this mechanism is not atomic; but it is deployable only through two steps.

4.4 O-rules and R-rules

Let us see how an inappropriate use of the o-rules combined with r-rules can cause unpleasant situations.

Assuming that `x-entity1` is the only co-owner of `b`:

```
## ACL of b ##
x-entity1: cdrwx
nobody:    -----
```

and `x-entity2` is the only co-owner of `c`:

```
## ACL of c ##
x-entity2: cdrwx
nobody:    -----
```

In this configuration no one is able change the object `c` except `x-entity2`. In fact, according to the `c` ACL table, if an entity different from `x-entity2`, tries to modify such ACL, the `others` entry will be picked by the decision algorithm. Therefore the requested operation will be denied according the rule expressed by the picked entry.

Even though `x-entity1` is unable to change directly the ACL of `c`, it can change the `b` one. Particularly it can add this new rule:

```
## ACL of b ##
x-entity1:  cdrwx
x-entity2: !-----
nobody:     -----
```

Hence the object `c` cannot be modified by anyone, unless x-entity1 removes the o-rule.

4.5 File-system configuration

Let us see how can be configured a simple file-system. In this example we are considering the structure shown in section 8.

These rules have to be defined:

- Entities must be able to read, write and traverse the objects guarded by its own guard (guarded by `entities`).
- Entities must not be able to manipulate the objects guarded by the guards of other entities (guarded by `entities`).
- Entities must not be able to manipulate the content directly guarded `root`.
- Entities must not be able to manipulate the content directly guarded by `entities`.
- Each entity can manipulate its `home` and the objects guarded by it.
- Each entity must be able to authenticate itself.

Below is shown a possible file-system configuration which is compliant with the rules just defined.

```
## root ACL ##
Admin:  cdrwx
nobody: --r-x
```

```
## entities ACL ##
Admin: cdrwx
nobody: --r-x

## home ACL ##
Admin:  cdrwx
nobody: --r-x

## x-entity guard ACL (guarded by entities) ##
## the same for all objects guarded by it##
Admin:    cdrwx
x-entity: --rwx
nobody:   -----

## home ACL of x-entity ##
Admin:    cdrwx
x-entity: cdrwx

## login ACL ##
Admin:  cdrwx
nobody: --rw-
```

	List	Alphabetic binary tree	Hash table
Entities Creation	Time: O(N) Space: O(N)	Time: O(Log N) Space: O(N)	Time: O(1) Space: O(M) If M >> N
Build entity tree	Time: O(1) Space: O(N) Amortized	Meaningless	Meaningless
ACL entry creation	Time: O(A) Space: O(A)	Time: O(Log A) Space: O(A)	Time: O(1) Space: O(M) If M >> A

Figure 11: Main model operations complexity. N: number of entities, A: number of the ACL entries, M: size of hash table.

ACL table \ Entities tree	List
List	Time: O(N * A) Space: O(N + A)
Alphabetic binary tree	Time: O(N * Log A) Space: O(N + A)

Figure 12: Matching operation between ACL list and entities tree complexity. N: number of entities, A: number of the ACL entries.

5 Performances

In this section some results which show the performances of the discussed ACL model are presented.

Since our purpose was not to create a high performance model, it will be reported how some improvements can be made.

A summary of complexities is also shown in figure 11 and figure 12.

Moreover each test was performed on a Intel(R) Core(TM) i5-2520M CPU @ 2.50 GHz machine using Linux Ubuntu 12.04.1 Precise Pangolin.

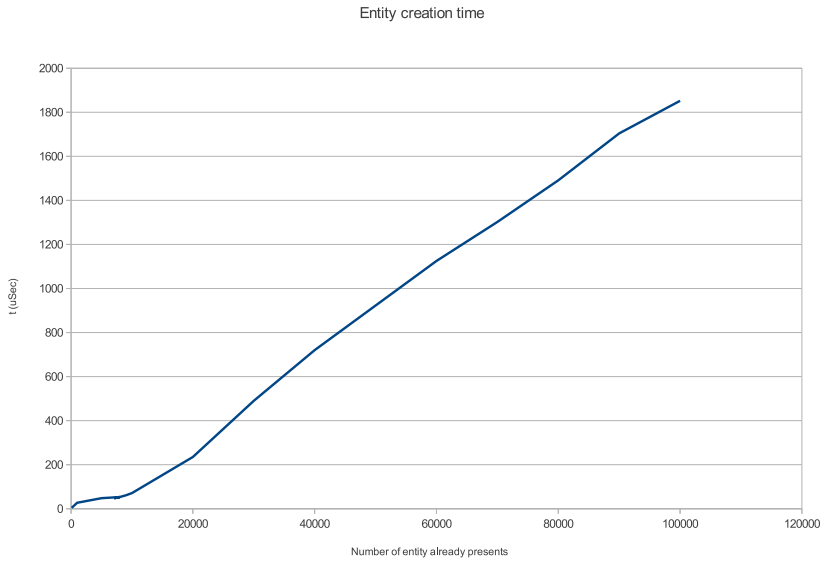

Figure 13: Entity creation process.

5.1 Add a new entity

When an administrator wants to create a new entity, it is necessary to check that such entity does not already exist.

Therefore is necessary to scan the entities list to check whether the new one can be created. Since a regular list has been used to keep track of the entities, this operation grows linearly with the number of entities present.

In figure 13 is shown a graphical representation of the necessary time to create a new entity (Y axis), as function of the number of entities already present in the system (X axis).

The experiment was conducted as follows: first, a file-system with a prefixed amount of entities was created; then, a new entity has been added recording the spent time to perform such operation. As we can see, this time grows linearly with the number of already existing entities.

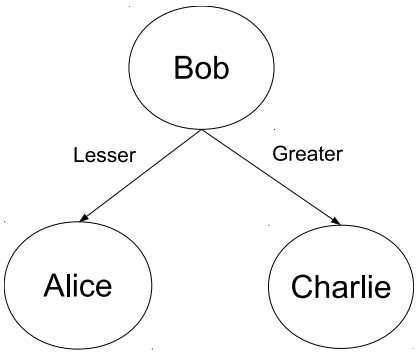

Figure 14: Alphabetic binary-tree example.

If N represents the amount of entities present in the system; by using a regular list, the used space to store such list and the time spent to update it, have complexity O(N).

A more clever approach may be to use an alphabetic ordered binary-tree in which entities are stored according their names. In such case the update/creation operation is performed with a complexity of O(logN), whereas the allocated space has still complexity O(N). In such tree each node reports the entity name, and can points to other two child nodes.

An example of alphabetic binary-tree is shown in figure 14.

In this case when an entity has to be retrieved, we can traverse the binary tree starting from the root and by choosing the branch to follow merely by performing a difference by the entity names. For example, let us suppose that we have to check whether the tree showed in figure 14 contains a node named "David". We have to start from the root and check whether "David" is alphabetic greater than the name contained in it. In this case "David" is greater than "Bob", hence the right branch is followed. At this point "David" is checked with "Charlie" and, again, "David" is alphabetically grater than "Charlie". Since "Charlie" is a leaf, the three does not contain any node named "David".

Finally a hash function could be used. In this case there would be two ways to implement this approach: by using open-hashing and by using close-hashing. In the first case, when an entity has to be added, a possible collision is managed by using a dynamic list of collided entities. Instead in the second case collisions can be managed by adding the new entry in the first available position in the hash table.

Complexity of this last approach depends of the hash table size. In fact if we represent such dimension as M, the operation of entity update/creation has complexity O(1) if M $>>$ N, whereas the allocated space is obviously linearly with M, hence O(M).

Since in a real scenarios it is correct to assume that not a so large amount of entities are managed by the same system; the binary-tree approach seems to reach the best trade-off between scalability and speed of execution.

Anyway, these operations are performed only when a new entity has to be created. Therefore such operations are not supposed to be performed so often in a real distributed file-system.

5.2 Build and scan entities trees

As reported in section 2.1, each entity in the system, owns a belong-to tree (or entities tree) where it is the root. Entities tree is the most important structure in the whole access control model because it is used every time the algorithm 2.2.1 must be performed.

In this test the performances obtained in building and scanning the entities trees are shown.

First, a certain amount of entities in relationship each other are set; then it will be shown how much time is spent in order to obtain and to scan the whole entity tree

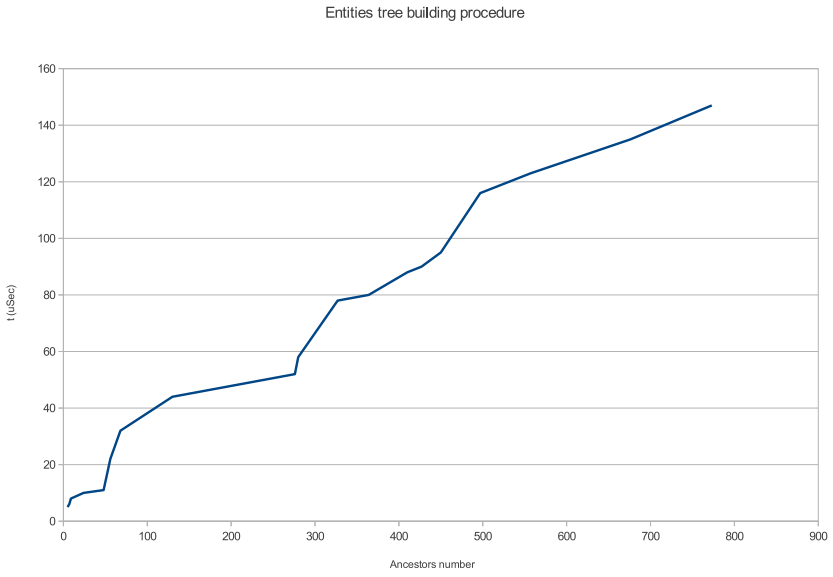

Figure 15: Entities tree building procedure.

of an entity.

Relationships among entities are created using a random function with a uniform distribution.

The ancestors value, reported as values of X axis in figure 15, has to be considered as the sum of direct parents and inherited both.

As we can see from figure 15, the complexity of the operation grows quite linearly with the number of ancestors which the selected entity has. Therefore if N is the number of the ancestors, the complexity is O(N) for what concern the spent time and the allocated space both.

Moreover, since all types of relationship nesting are allowed, cases of infinite loop may occur. These cases are avoided merely by using an integer which marks every entity visited during the scan procedure.

In order to find the most accurate match for an entity, an appropriate sorted list of ancestor is necessary. This list has to be sorted according the priority of such

ancestors with the main entity (root of entities tree) as explained in 2.1.

When we have to find the most specific rule in an ACL, it will be necessary to scan the entities list, element by element, until a valid match is not retrieved with the ACL list, or the entities list is over. These are the cases in which entities lists has to be scanned.

There is no linear dependency in building entities trees, since such operation has been implemented with constant time complexity (or O(1)). This is a very good result since a heavy nesting among entity relationships, does not introduce additional complexity. This goal was achieved merely using for each entity a list of pointers which refer to the entities which it directly belongs to. In other words a list of pointers to the first level ancestors.

During the checking algorithm it is only necessary to follow such pointers iteratively. A new pointer is added in an entity list, if such entity holds a new relationship with another entity.

Therefore time complexity of building entities tree is time constant (O(1)) and allocated space large as the amount of first level ancestors. The only linear dependency is obtained only when we have to scan it. This complexity is the lowest possible.

Of course, whether in following pointers of a list, a swapped entity is found, we have first to load it in order to retrieve its entities list. In these cases time spent to scan an entities list will grow.

It is important to say that is useless to use a real tree structure to represent an entities tree. In fact the only cases in which we have to access to the entities tree are when we have to find a rule for an entity. In these cases we have to check first the principal entity (root), then the first level ancestors, then the second ones and so on. Therefore by using a tree we would check first the first tree level, then the second one and so on until the last tree level. This behavior is exactly the same which we can

obtain by using a classic list. Therefore, whether the word "tree" is used to refer to the ancestors structure, this has been implemented as a simple list.

5.3 Add an ACL entry

The procedure shown in section 5.1 has been adopted also when a new ACL entry has to be added in an ACL list. Before to add a new entry is necessary check if there is another entry which represents the same entity named in the new rule. Therefore a list of entries has to be visited. This test case results to be exactly the same shown in section 5.1.

In this case anyway it would be necessary make some changes in order to improve the performances, since this operation could be more frequently than the one explained in section 5.1.

For this reason a really good way to represent an ACL table, would be the one in which the rules are stored in a binary-tree arranged in alphabetic order.

If the number or rules contained in an ACL table is A, then the allocated memory would have linear complexity O(A). Instead the necessary time to update such table would have logarithmic complexity, hence O(log A).

Finally, also in this case a hash function could be used, but for the same reasons expressed in 5.1 this approach is not the best.

5.4 Collection of entity rules

As reported in section 3.4, rules are collected during the entity walk operation.

The algorithm showed in 2.2.1 states that the rule picked from an ACL is the most specific for an entity. Therefore if an entity is requesting to perform an action, we would have to scan its whole entities tree, level by level, in finding the first entity

which is contained in the ACL table (if there is one).

Therefore cases in which is necessary to scroll these two structures entirely (ACL table and entities tree both) represents the worst cases.

In this test both lists (also entities trees are lists as it has been said in section 5.2) are been created with the same number of elements. Moreover, in order to show the worst case, it has been positioned the only entity which is present in both lists, at the end of both.

As we can figure, complexity of this operation is quite quadratic; hence if N is the number of ancestors of an entity and A the amount of rules contained in an ACL table, time complexity of this operation is O(N*A).

Figure 16 shows the well-known quadratic shape.

This result could be improved using again a binary-tree. In fact, whereas the entities list must be scanned linearly element by element, an ACL table can be represented as a binary-tree (according what has been said in section 5.3). In this way, for each element contained in the entities tree, the search operation is performed with complexity O(Log A). Hence for finding a match for an entity in an ACL table would be overall O(N * Log A).

5.5 Scan an ACL table

Since ACL rules are stored in entities stack-caches (see section 3.4), entity matching actually is to pick the last value from its cache. Therefore complexity of these operations is constant (O(1)).

5.6 Scan a delegations list

As showed in section 3.4, delegations are not collected during the walk procedure.

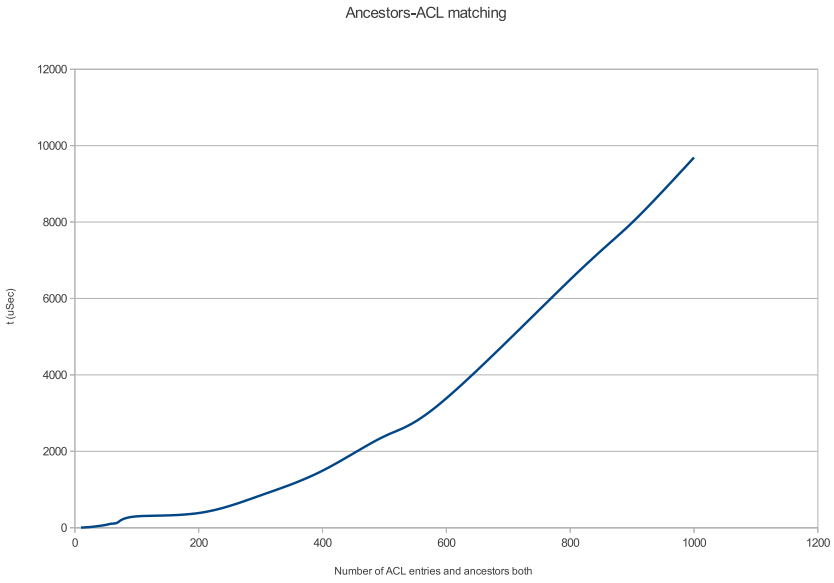

Figure 16: Ancestors ACL matching algorithm.

Therefore when it is necessary to check the delegations, the same algorithm showed in section 5.4 is followed. Therefore the same considerations are also valid in this case.

6 Problem experienced and solutions proposed

In this paragraph some conceptual problems that are been experienced, and their solutions, are shown.

6.1 ACL definition

First problem was to define what exactly ACL has to be and where physically stores it. An ACL is a set of rules which regulates the accesses to an object and to its attributes; so an ACL has to be tied to the object it represents.

Problems have been posed were:

- if an ACL is put in a file which links to the object represented by the ACL; who establishes the rules to access to the first file?
- If an ACL is put inside the object which it represents; who establishes the rules to access to the file part where the ACL is stored?

In the proposed model ACLs are stored among file attributes as showed in figure 17.

Moreover such attribute is treated as a special one since no one has to be able to delete it, and only the ACL co-owners has to be able to change it.

In order to modify an ACL, an entity, writes on the ACL attributes the changes and only if the request is compliant with the ACL rules it will take effect.

This operations are made automatically by using the command `setfacl`, as showed in 3.7.2.

6.2 Delegation definition

A delegation behaves exactly like a temporary ACL rule; for this reason delegations could be implemented modifying temporarily an ACL table, and storing there the

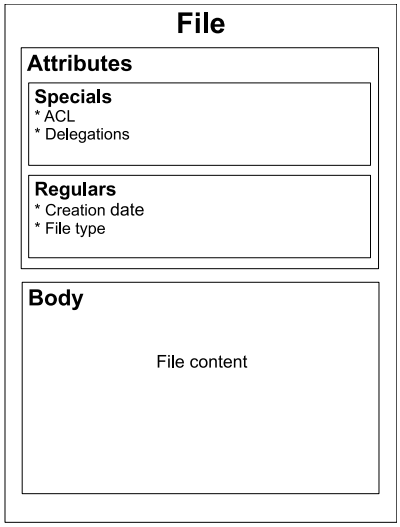

Figure 17: ACL is placed among specials file attributes.

delegations rules. Even though this procedure is useful; it makes an exception. In fact, according an ACL, only the co-owners are allowed to modify the rules contained in it; but anyone who holds at least one valid permission must be able to delegate it. So if we want to place a delegation among the ACL rules we have to break this rule, otherwise only the co-owners would be able to issue valid delegations.

Therefore a new file, called `proxies`, has been introduced in every `entity` guard (figure 8). Such file contains the delegation issued by the entity represented by the guard. This approach does no introduce any exceptions.

Another problem about the delegation mechanism was to decide where store them once they were valid. For the same reasons showed in section 6.1, the delegations are stored among the special attributes (see figure 17).

A new delegation will become valid when the system will approve it. The system approves a delegation only if the permissions expressed in it, are compliant with the issuer rights on the file-system object reported in the delegation.

Unlike from the ACL case, this mechanism is completely hidden from the user point of view, entities are aware only that in order to create a new delegation they

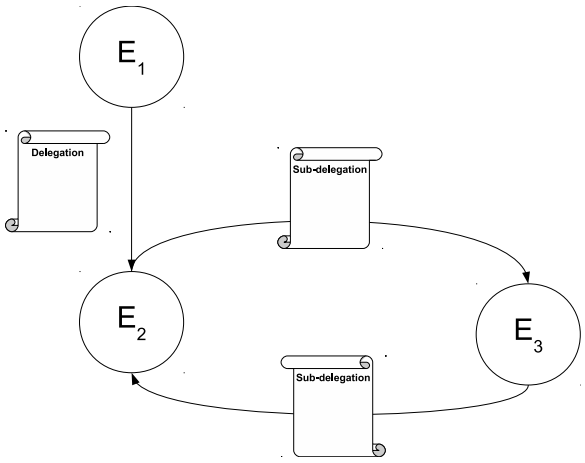

Figure 18: Sub-delegation loop.

have to write it in their `proxies` file.

If the delegation is not valid no errors are reported, the solely consequence is that such delegation will not be used in the decision algorithm.

6.3 Sub-delegation

Another question to define was about the possibility for a delegatee to delegate another entities using its delegations.

Without define any rule some unpleasant situations may occur. For example, assuming that there are no limits about the delegations usage; a delegated entity may use its delegations to sub-delegates another entity which, in turn, it may sub-delegate the first one and go on; making thus the delegation expiration time limitless.

As showed in figure 18, E_1 issues a delegation for E_2 which could sub-delegates E_3 using an expiration time longer than the one expressed in the original delegation. In turn E_3 could sub-delegates E_2 using an expiration time longer than the one expressed in the previous sub-delegation; and go on.

Therefore some rules have to be defined. Particularly two important aspects of a

delegation have to be evaluated:

- Sub-delegations lifetime.
- Trust chain depth.

The first parameter states how much time a sub-delegation can be considered valid; whereas the second one states how long the sub-delegation chain can be.

Users can decide to express (or not) the value of the second parameter, which is left unset by default (hence a limitless length of trust chain). This value merely represents the maximum number of time which a delegation can be used in order to delegate other entities, before to be considered no more valid to delegate.

Instead, for what concern the timing question (first parameter), an expiration time has to be mandatory expressed in every delegation. Moreover entities are not allowed to sub-delegate other entities for an expiration time beyond the one indicated in the delegation used to issue the new one. This rule is transitive in all the trust chain. For example, given a chain of delegation`a->b->c...k-1->k`, `b` cannot have an expiration time beyond the one expressed in `a`, as well as `c` cannot have an expiration time beyond the one expressed in `d` and, clearly, same reasoning up to `k`. This avoids delegations endless cases.

6.4 Co-owners

Since that more than one owner is allowed in a file ACL, may happen that they want to remove each other from the ACL table. If this would be possible, the object could remain without anyone able to change its access rules. The first followed approach was to define a main-owner which cannot be directly removed unless it gave its main-ownership to another entity. Later it was decided that this way was slightly intricate; so it has been decided to deny the removal to the last named co-owner in the ACL.

Note that if an ACL table does not contain the entry `nobody` (or `others`) the co-owners removal does not imply any problems, since the rules of its guard are automatically inherited. In fact, if an entity wants to modify an object ACL in which no co-owners are named, according the decision algorithm expressed in 2.2.1, the rules of the guard are taken in account. Instead if the entry `nobody` (or `others`) is present, it inhibit the rules inheritance. In this last case if neither `others` nor `nobody` is able to manage the object ACL and no co-owners are present in such ACL, no entity will be able to change the object rules. Hence the file-system administrator has to intervene to solve this deadlock.

By assuring that at least one co-owner is alway named in an ACL table, we are able to avoid all these problems.

6.5 Entities relationships

As it has been said in 2.2.1, entities are able to use their ancestors rights during the decision algorithm, if no more specific entry is matched in an object ACL.

This suggests that a protocol which regulates how two entities can stipulate a new relationship has to be defined.

The following approaches were taken in consideration:

- Let the administrator decide what relationships create.
- Create a new relationship when an entity request it.
- Create a new relationship when both part involved agreed.

The first one is the most monitored, but also the most rigid since when an administrator has defined a relationship, this cannot be modified by anyone but it. Moreover

anytime which two users want to create a new relationship they have to ask the permission to the system administrator, which could not be available. Therefore this solution could introduce a huge wasting of time.

The second one, Even though is the most efficient in terms of execution time, it is the less secure according the role of the entity which requests to create a new relationship. Indeed two different parts can make such request: the father and the child. In order to better understand, considering a general relationship `entity2` $\in$ `entity1`, we refer as *the father* `entity1` and as *the child* `entity2`. If the approach used foresees that the requester is the child, an entity may recklessly belongs to every entities it wants to; having so theoretically access to every objects it wants to. This solution is therefore the less secure. The other case presents the opposite drawback. If the requester is the father could happen that an entity wants to make a new relationship with another one, merely to harm it. For example, assuming that there is an ACL table that states that `entity1` are not allowed to read a file-system object, and does not say anything about `entity2`. In this case `entity1` may want to create the new relationship `entity2` $\in$ `entity1` just to harm `entity2`. In fact, during the access control, if no rules for `entity2` are retrieved, the parents ones are checked.

Finally the third one is a good trade-off between flexibility and security. For these reason this one has been implemented.

The mechanism in which two entities create a new relationship is expressed in 3.3.

6.6 Not overridable rules behavior

Not overridable rules allow to configure the system in few steps and to safely control what users are able to do.

These rules cannot be operative rules, otherwise it would be impossible to express rules like: “everyone is be able to” without implicitly define “everyone can do”.

For example, assuming that a rule which allows to everyone who is logged to be able to be a co-owner (on a given object) has to be defined. As we saw in section 4.2.4 the rule to add is:

```
others: !cdrwx
```

This is possible only because this rule is not effective, but it merely represents an upper-bound limit. If it were a real rule its meaning would be “everyone who is logged is a co-owner”.

This model of rules, even though simple to understand, presented a lot of problem. In fact their priority with the others rules had to be well defined.

A trivial case may be one in which, in the same object, there is a r-rule which names directly an entity, and an o-rule which names one of its ancestors.

Since an o-rule has priority on the r-rules, but the r-rule is more specific, what should be the right rule to follow? Just one (and in these cases which one?), or both combined?

These situations are solved as showed in 2.2.2. In other words, we search in every ACL both type of rules separately, by applying the algorithm expressed in 2.2.1.

6.7 Not overridable rules and delegations

Another question concerning the o-rules was about what it would happen if an entity delegates another one to perform a given action which, according some o-rules, could not perform. This scenario is quite different to the one reported in section 6.6. Indeed in this case the entity who has requested to perform an operation, would act in behalf

of another entity which is able to do it. This situation is not trivial.

In order to understand how this can be very tricky, let us see a theoretical example.

Assuming that path `a/b/.../f` exist, and that the ACLs of the guard `a` and `b` respectively are:

```
## ACL of a ##
x-entity1: !c-rwx
```

```
## ACL of b ##
x-entity1: c-rwx
```

According these tables `x-entity1` is unable to delete objects beyond `a`, but it is able to decide who can do it thanks to the co-owner bit. Therefore `x-entity1` can write a new rule in the ACL of `b`, naming `x-entity2` in such way:

```
## ACL of b ##
x-entity1: c-rwx
x-entity2: -drwx
```

Besides having rights to delete objects, `x-entity2` is now also able to delegate other entities to do it in its behalf. Particularly `x-entity2` can delegate `x-entity1`, overcoming the o-rule presents in `a`.

Even though this does not create inconsistencies, since `x-entity1` acts in behalf of `x-entity2` (which is allowed to delete file-system objects) this has been possible solely thanks to `x-entity1`.

In order to avoid all these tricks which could override the o-rules, it has been decided to give higher priority to the o-rules rather than the delegations.

6.8 Authentication management

Another question we faced was about the way in which entities can log onto the system. Ideally each entity could own an object in which writes its credentials to authenticate itself. Such an object could be placed in the guard which names the entity, contained in the `entities` guard.

Moreover, in order to authenticate an entity, the object has to have the write permission for `nobody`, since every entity before to be authenticated is unknown from the server point of view. This fact introduces a huge security issue.

Let us suppose that a `x-entity` is writing on its `login` object in order to authenticate itself. Since `nobody` can write on such object, a malicious entity could write a sort of "null" letter at the end of the file, a moment before which `x-entity` sends the request, authenticating itself as `x-entity`. The system is not able to prevent these situations since each entity, before to be authenticated, is `nobody`.

Moreover when an `x-entity` is authenticated, the object content must be deleted, otherwise every entity by writing a "null" letter would be able to authenticate itself onto the system as `x-entity`. This is possible merely because `nobody` has to be able to write in the `login` object.

A possible solution for all these problems may be to use a common object for all the entities, where each entity has to take its lock before start writing its credentials. In this case only an entity at time takes the access to the object, avoiding all the problems explained up to now. Unfortunately this approach implies new drawbacks. Indeed if an entity session crashes, the object is left locked; making impossible to authenticate other entities. Furthermore a malicious entity may block on purpose such object just to harm the users which want to authenticate themselves onto the system.

The solution adopted plans to use a special file together with a session buffer, in which the entities write their credentials.

From the user point of view, when it wants to authenticate itself, it has to write on such file; but, when this happen, the content is automatically redirected to its session buffer. Such file is not really present into the file-system, but is merely an interface which can be used by the users to authenticate themselves.

In this approach entities are unable to create dead-lock situations and, since actually they do not write on the same object, all the problems highlighted are avoided.

Part IV

Comparison with other models

The proposed ACM design takes cue from the other well-known models, particularly from the ACL POSIX one.

In this section are going to be discussed the differences between our model and two other well-known models for the accesses control, which have mostly inspired the presented one.

Throughout this part, for simplicity of reading, we refer to an entity which contains other entities as a *group*, and to a single entity as a *user*.

7 UNIX standard and ACL POSIX

The traditional UNIX file-system object permission model defines three classes of users: *owner*, *group* and *others*. Each of these classes are associated with a set of permissions which define the rules to access to the object. Available permissions are: read (r), write (w) and execution/traversal (x).

In such model the owner entry represents the object owner permissions, group represents the owner group permissions (*owning group*) and finally *others* represents the permissions held by everyone else in the system.

A POSIX ACL table is a collection of rules, which represent a user (or a group) and its permissions on a file-system object. Each one of these three classes of users is represented by an entry in such table; whereas permissions for additional users (groups), called named users (named groups), occupy additional ACL entries.

A POSIX ACL may contain any number of named users (or named groups) entries,

which are automatically assigned to the *group class*.

The meaning of the permissions expressed by the *group class* is merely to bound the permission which every entry, assigned to this *class*, can own. Therefore permissions contained in this class act like a *mask*: every named entry/group permissions are masked used the permissions indicated by the *group class* itself.

This typ of ACL is called *access* ACL and defines the current access permissions of a file-system object.

A second type of ACL, called *default* ACL, can be used in order to implement the inheritance concept. This type of ACL defines the permissions a file-system object inherits from its parent directory at the time of its creation. This type of ACL plays no direct role in access checks, but it is merely used when a new file-system object is created, in order to collect a set of initial rules. Furthermore, only directories can be associated with a *default* ACL.

7.1 Comparison with Pepys ACM

The ACM proposed in this document extends the concept of mask (as defined by the *group class* in the POSIX ACL) by introducing the *o-rule* concept. In such way it is possible to decide whether (and in what file-system position) to apply a certain restriction, and for which entity.

Moreover whereas the file owner in a POSIX ACL cannot be changed (it is who has created the file for all the file lifetime), in our model it is possible to transfer the ownership merely by adding a new entry which contains the co-owner bit set in the permission set.

Furthermore, in the presented model, it is possible to let an object dynamically inherit the rules of its guard, merely by not adding the *nobody* (and in case *oth-*

ers) entry in the object ACL. Therefore if the parent guard rules change, they are automatically inherited from the guarded objects.

Finally, in the POSIX ACL, the presence of the *others* entry is mandatory; in our model it is not.

7.2 Access check algorithms

The algorithms followed by these two models can be divided in two common steps:

1. Select the ACL entry which matches most closely the user who requested the operation.

2. Check if the matching entry contains sufficient permissions to let the user proceed.

In the POSIX ACL model, in the first step, ACL entries are looked in the following order: owner, named users, named groups, others.

Moreover if a user is a member in more than one group named in a POSIX ACL, and if some of them contains the necessary permission to grant the requested action, one is picked (the result is the same regardless of which one is picked). If none of the matching groups entries contains the requested permission the access would be denied regardless of which entry is picked. If no entry is found for the requesting user the *others* permissions entry determines the access to the object.

In our model, entries in an ACL table are looked in the following order: entities, others, nobody.

When a user want to perform an action on a file-system object, if its name is not directly reported in the object ACL rules, the algorithm picks the entry which has highest priority (see section 2.1). If there are not any entries which refer (directly or indirectly) to the user, the rules of the guard above are considered.

Therefore, the ACL POSIX algorithm tries to find an entry which could grant the access to the requesting user, whereas in our model the most accurate entry is searched to determinate the access.

Finally in both of these two algorithms, when an entry is picked, a permissions mask is applied to retrieve the real user permission. However, whereas in POSIX ACLs such mask is contained in the same ACL table, and fixed by the file owner; in our model the mask is determined also by the ACLs of the guards up in the file-system hierarchy. Note that in our case masks are given by a collection of o-rules.

8 NTFS

In the NTFS model, the basic permissions which can be assigned to files are: Full Control, Modify, Read & Execute, Read, and Write. Whereas, for folder the available permissions are: Full Control, Modify, Read & Execute, List Folder Contents, Read, and Write.

Particularly for what concern folders, Read permission make a user able to view and to list files and sub-folders contained in it; the Write permission, accordingly, allow to add files and sub-folders. Read & Execute permission permits viewing and listing of files and sub-folders as well as executing of files; inherited by files and folders. List Folder Content makes a user able to view and to list files and sub-folders, as well as executing of files; this right is inherited by files and folders. Modify permission allow to read and to write files and sub-folders; moreover it allows deletion of the folder. Finally Full Control allows to read, to write, to change, and to delete files and sub-folders .

Instead, for what concern the files, Read permission allows to a user to read the file content, the Write one to write it. Read & Execute permission permits viewing

and accessing of the file's contents as well as executing of the file. Thanks to the Modify permission a user is able to read, to write and to delete a file. Finally with the Full Control permission, a user can read, write, change and delete the file.

Moreover, in NTFS file-system, if a user owns full control permission over a folder, it can delete files contained in it, regardless of the permission on such files.

Actions that users can perform are based on the sum of all the permissions assigned to the user and to all the groups the user belongs to ([29]). For example, if a user owns the Read access to a given files, and a group in which it belongs to has the Modify access on the same file, the user will have Modify access. This behavior is inherited, hence if a parent group belongs to the administrator group, the user will own Full Control permission. If no permissions are retrieved for a requesting user, the access on a file-system object is denied.

Since also in this model a user can own a list of permission, the ACL approach is used.

Another interesting feature is about the usage of two family of rights: Allow and Deny.

When establishing permissions on a file-system object, a user has to specify whether the entry should have access (Allow) or not (Deny) to such object. Since already the lack of a permission match, for a particular user, is considered as "permission denied", cases in which Deny permissions are necessary are not so common. Finally, the Allow and Deny permissions inherit down through the file-system structure.

8.1 Comparison with Pepys ACM

Pepys ACM model presents some little common features with NTFS model.

For example, the idea to use an independent permission to establish if an entity is

allowed to delete file-system objects, was taken by studying the NTFS model. Thanks to this approach is possible, for example, to define entities which can write defined files but which they do not have the faculty to delete them. This is not possible in file-systems UNIX-based, because the faculty of delete file-system object is bound to the write permission.

In our model the execution permission is not present. This fact does not apply any limitation to the available operations since the file-system objects are remotely stored (Pepys file-system is a distributed file-system); hence the execution permission (alone) would be useless. According to this, if we cut off the execution among the NTFS operation, our permissions set and the NTFS one are pretty the same.

Another common feature is the algorithm outcome when no rights, for a given user, are retrieved. In fact, in both model, the access is denied. Furthermore the rights inheritance in both models are very similar.

Finally there is just one significant difference between these two model and it is the way to apply exceptions. Whereas in the NTFS case if a user owns Full Control permission (on a folder) it is able to delete the whole content, int our model is possible to add exceptions merely by adding the rules in the object ACL contained in the folder, or using the Pepys semantic, guarded by the guard.

Our model appears to be more flexible and manageable than the NTFS one. For example thanks to the Full Control permission, the actions performed by user, may become uncontrollable. This can be even more dangerous if such permission is assigned to a group. Instead in our model is possible to add exceptions in any case for any file-system objects/users.

Part V

Conclusions and Future Work

An access-control model for the Pepys Internet-wide distributed file-system has been proposed, showing the characteristics of its design. The proposed model takes into account the basic principles behind the well-known POSIX ACL standard and other widely used file-systems, enriching the model with characteristics that are inspired to the general principles of the Pepys distributed file-system.

This model aims to cover the main aspects which a distributed file-system access control should have, that is: authentication, authorization, granularity, autonomous delegation and revocation; as properly expressed in [17].

This document provided also a few notes on how the model has been implemented in a Linux port of the Pepys current code base.

Possible future work on the topic include: integration of Pepys and particularly of the current authentication mechanism with properly designed cryptographic extensions to the ΠP protocol and possibly optimize the most recurrently used code paths improving performances.

Part VI

Special thanks

The presented work has been possible thanks to a lot of people beside me; in this section I wish to thank all of them for the opportunity they gave to me.

Particularly I wish to thank my family, without its support nothing of this would have been possible. I wish to thank them also for the trust they gave to me and for helping me financially during my stay in Dublin.

I wish to thank Mr. Giuseppe Lipari, from the university of Pisa, which introduced me to Mr. Cucinotta, in order to have a working experience abroad.

A really big thank goes to Mr. Tommaso Cucinotta which was my tutor during the whole period spent in Bell-Labs Ireland; for his precious help and for his kindness. I wish to thank him also for the opportunity he gave to me to present this work in the international Plan 9 workshop during my internship period.

I take this opportunity to also thank Mr. Sape Mullender (Director of the Network Systems Laboratory at Bell-Labs Antwerp) for his help during the porting phase, Mrs. Julie Byrne (Executive Director Bell Labs Ireland and UK at Alcatel-Lucent), Mrs. Alessandra Sala and Mr. Ivan Bedini.

I want to thank Mr. Holger Claussen (Head of the Autonomous Networks Department at Bell Labs Ireland) to let me to be part of his great team, and all of its not-yet-mentioned components: Mr. Davide Cherubini, Mr. Francke Franck, Mr. Dirk Hasselbalch, Mr. Eric Jul, Mr. David Lopez-Perez, Mr. Frank Mullany, Mr. Florian Pivit and Mr. Vijay Venkateswaran. A big thanks to all of them to had taught me a lot of interesting things.

I wish to thank Mr. Gianluca Dini, from university of Pisa, for his help and for

agreeing to be my advisor in this work.

I want to thank all my friends which are always been by my side and, particularly, a big thanks go to my girlfriend, Giulia Ponzetta, which has supported (and endured) me during these, not always easy, university years.

Last big thank go to my, no more present, grandmother; without her help perhaps I would never became an engineer.

References

[1] *Operating Systems: Three Easy Pieces*, chapter 48, page 702. Number 9781105979125. Edition 0.5 edition.

[2] *IEEE Std 1003.1-2001, Open Group Technical Standard–Standard for Information Technology–Portable Operating System Interface (POSIX)*, 2001.

[3] D. Elliott Bell, L. J LaPadula, and United States. Air Force. Systems Command. Electronic Systems Division. *Secure computer systems : a mathematical model / D. Bell, J. LaPadula.* Springfield, Ca. : National Technical Information Service, 1973. Reproduction. Originally issued: Bedford, Mass. : Mitre Corporation, 1973.

[4] D.F.C. Brewer and M.J. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206 –214, may 1989.

[5] Digital Equipment Corporation. Decnet-plus foropenvms introduction and user's guide. Technical report, nov 1996.

[6] Goerge Coulouris, Jean Dollimore, and Tim Kindberg, editors. *Distributed Systems: Concepts and Design*, chapter Amoeba. Addison-Wesley, 1994.

[7] Russ Cox, Eric Grosse, Rob Pike, David L. Presotto, and Sean Quinlan. Security in plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, Berkeley, CA, USA, 2002. USENIX Association.

[8] Jason Crampton and Hemanth Khambhammettu. Delegation in role-based access control. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Computer*

Security – ESORICS 2006, volume 4189 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin Heidelberg, 2006.

[9] Tommaso Cucinotta, Nilo Redini, and Gianluca Dini. Access control for the pepys internet-wide file-system. In *Proceedings of the 7th International Workshop on Plan 9*, Blanchardstown Business and Techonology Park, Snugborough Road, Dublin 15, Nov 2012. Bell-labs.

[10] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. RFC 2617, jul 1999.

[11] Andreas Grünbacher. Posix access control list on linux. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[12] Apple Inc. Apple filing protocol concepts.

[13] Michael Kaminsky, George Savvides, David Mazieres, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 60–73, New York, NY, USA, 2003. ACM.

[14] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, November 1992.

[15] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National*

Information Systems Security Conference, pages 303–314, Crystal City, Virginia, 1998.

[16] Microsoft. Microsoft smb protocol and cifs protocol overview, 2009.

[17] Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos Keromytis, and Sotiris Ioannidis. Decentralized access control in distributed file systems. *ACM Comput. Surv.*, 40(3):10:1–10:30, August 2008.

[18] Sape J. Mullender and Andrew S. Tanenbaum. Protection and resource control in distributed operating systems. *Computer Networks*, 8(5-6):421–432, 1984.

[19] E. Barka nd R. Sandhu. A role-based delegation model and some extensions. In *In Proceedings of 16th Annual Computer Security Application Conference*, pages 11–15, Sheraton New Orleans, dec 2000.

[20] Novell. Netware core protocols.

[21] Herman C. Rao and Larry L. Peterson. Accessing files in an internet: The jade file system. *IEEE Trans. Softw. Eng.*, 19(6):613–624, June 1993.

[22] J. Sacha, J. Napper, S. Mullender, and J. McKie. Osprey: Operating system for predictable clouds. In *Proceedings of Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1 –6, june 2012.

[23] Pierangela Samarati and Sabrina de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of security analysis and design (tutorial lectures)*, pages 137–196. Springer Verlag, 2001.

[24] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Innovations in internetworking. chapter Design and implementation of the Sun network filesystem, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988.

[25] Russel Sandberg. The sun network file system: Design, implementation and experience. Technical report, in Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition, 1986.

[26] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The nist model for role-based access control: towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 47–63, New York, NY, USA, 2000. ACM.

[27] S. Shepler, Storspeed, M. Eisler, and D. Noveck. Network file system (nfs) version 4 minor version 1 protocol. RFC 5661, jan 2010.

[28] Guido Socher. File access permissions. 2000.

[29] William R. Stanek. File and folder permissions. In *Microsoft Windows 2000 Administrator's Pocket Consultant*, page Chapter 13, 2002.

[30] SNIA: storage Networking Industry Association. Common internet file system (cifs) technical reference. Technical report, SNIA, jan 2002.

[31] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. Webos: operating system services for wide area applications. In *Proocedings of High Performance Distributed Computing, 1998. The Seventh International Symposium on*, pages 52 –63, jul 1998.

[32] K.G. Walter, W.F. Ogden, W.C. Rounds, F.T. Bradshaw, S.R. Ames, CASE WESTERN RESERVE UNIV Cleveland Ohio dept of computing, and infor-

mation sciences. *Primitive Models for Computer Security.* Defense Technical Information Center, 1974.

[33] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A rule-based framework for role-based delegation and revocation. *ACM Trans. Inf. Syst. Secur.*, 6(3):404–441, August 2003.