Access Control for the Pepys Internet-wide File-System

Tommaso Cucinotta, Nilo Redini
Bell Laboratories, Alcatel-Lucent Ireland

Gianluca Dini
University of Pisa, Italy

October 31, 2012

Abstract

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

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`. In the Pepys design, the traditional distinction among files and folders is replaced by the ideas that a collection of files (called *objects*) reside in the file-system. 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 3.

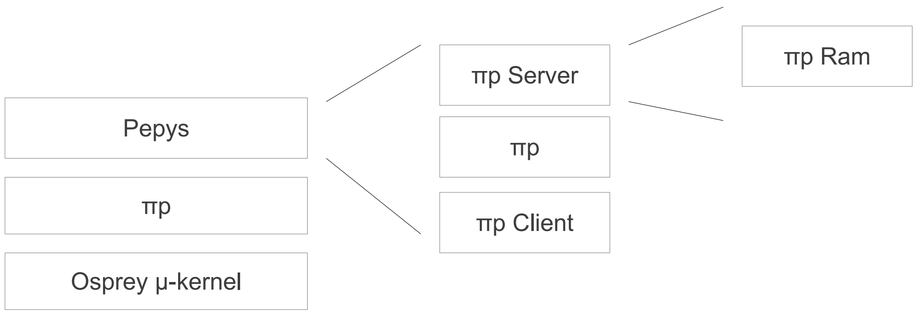

Figure 1.1: Pepys components.

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.

Furthermore, Pepys uses a new transport protocol (called ΠP) in order to minimize the round-trip message exchanges, between a client and a server, necessary to perform file transfer operations. The protocol allows to send, to the server, multiple consecutive requests in a single packet.

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.

Pepys file-system is currently implemented on top of a new operating system called *Osprey* [9] (see Figure 1.1), providing an alternative approach to cloud computing, and specifically aiming to improve latency and predictability of cloud applications and support for mobility. A key component in the overall architecture is the ΠP protocol, supporting all Pepys operations, including various interactions with the Osprey kernel itself.

As shown in Figure 1.1, the original Pepys server we modified included ΠP Ram, basically an in-RAM file-system. As explained in 4.1, this has been extended to keep files on a Linux (and generally POSIX) file-system, and to support our new AC model.

1.1 Paper Organization

The reminder of this paper is organized as follows. In Section 2, we put our work in relationship with related existing works in the literature. In Section 3, we present the Access-Control Model (ACM) we designed for Pepys, highlighting the most important design choices. Section 4 provides some implementation and further architectural details. Finally, in Section 5 we describe possible future work we plan to do on the topic.

2 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, and specifically:

- Unix File-System permissions [11] and Linux extensions [13]
- New Technology File-System (NTFS) permissions [12]
- POSIX Access Control Lists (ACLs) [1, 5]
- Role-Based Access Control (RBAC) [10]
- Discretionary Access Control (DAC) [7]
- Mandatory Access Control (MAC) [7]
- HTTP authentication mechanism [4]

Our work was greatly inspired to the POSIX Access Control Lists (ACLs) [1, 5]. POSIX ACLs overcome some of the limitations of the old UNIX file-system [11], allowing for the definition of multiple per-user and per-group rules, providing a great liberty of flexibility in expressing access-control rules. The access-control model proposed in this paper 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 3 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 [11]. 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 [12], 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 [10] models.

Our model design allows users to manage their own files permissions, allowing for a completely discretionary access-control, as found in DAC [7] 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 [7] models, and taking inspiration from similar characteristic available in in NTFS.

Our implementation did not address comprehensively *authentication*, yet. However, a basic authentication mechanism has been realized, taking inspiration from HTTP-Auth [4], 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 [3]. Furthermore, we support a primitive mechanism for *delegation [6]* of authority through off-line delegation certificates resembling Amoeba *capability lists* [8, 2].

Various other access-control models for file-systems have been proposed in the literature, such as the WebOS [14] work, including a mechanism allowing entities to delegate other entities in order to act on their behalf on a set of defined file-system objects, or others. A comprehensive list of such works is out of the scope of the present paper.

3 Access Control in Pepys

One of the basic concepts behind the Pepys access-control model design is the one to create an environment in which:

- the traditional distinction between users and groups is replaced by a unified vision of such entities;
- AC 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;
- each user is free to define the access control rules for its own objects, in the most flexible way possible;
- 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.

3.1 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.

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 overriden by more specific rules.

There are no limitations for the nesting level of the belong-to relationship, which is to be considered a transitive relationship. 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 3.1).

Moreover, as we will see, an entity authentication is not mandatory: an entity can decide whether or not to authenticate itself into the system.

Two system-level entities are always defined in the system, called `others` and `nobody`. Each entity defined in the system belongs implicitly to `others`, but only in the weakest possible sense (see Section 3.2.1). The `others` entity 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. The `nobody` entity may be conveniently used in AC rules to refer to any unauthenticated user. Also, is the system implicitly considers that `others` belongs to `nobody`, as shown in Figure 3.1. The purpose of these two entities is further detailed in Section 3.3.

Finally, since nesting relationships can be arbitrarily added by users, loops are possible in the belong-to tree. Such a situation, albeit unusual, is still handled by the implementation consistently.

3.2 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.

Each ACL can have one or more *co-owners*, which can manage the rules in the 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.

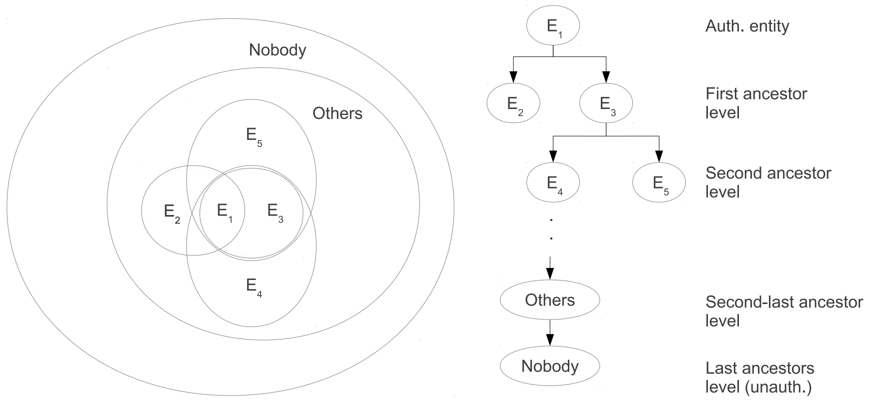

Figure 3.1: Belong-to relationship tree, rooted at a generic entity E_1.

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 file), and any other further object down the containment/guard hierarchy of objects (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 rule in an object (both a regular rule or an o-rule) 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 instead to grant access to any user connected to the system, but not having 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 inspired to traditional UNIX file-systems: read and write of files, traversability of guards, ACL management (co-ownership). However, this tentative set of permissions can easily be extended to more complex permissions or permission set (e.g., adding a delete permission or others, as found on NTFS file-systems). 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.

3.2.1 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, 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) if there is a match, 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.;
   (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.

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.

3.2.2 Delegation

Entities can delegate others entities 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 delegee, a set of permissions assigned to the delegee and an expiration date. Clearly, the permissions granted by delegation cannot be higher than the ones held by the delegator on the object.

Two kinds of delegation are possible:

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 delegee, its permissions and the file-system object on which the delegation is applied. In the second method the delegator issues a signed delegation to the delegee which, when it wants to perform an action on behalf of the delegator, will present it to the system.

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

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

3.3 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.

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.

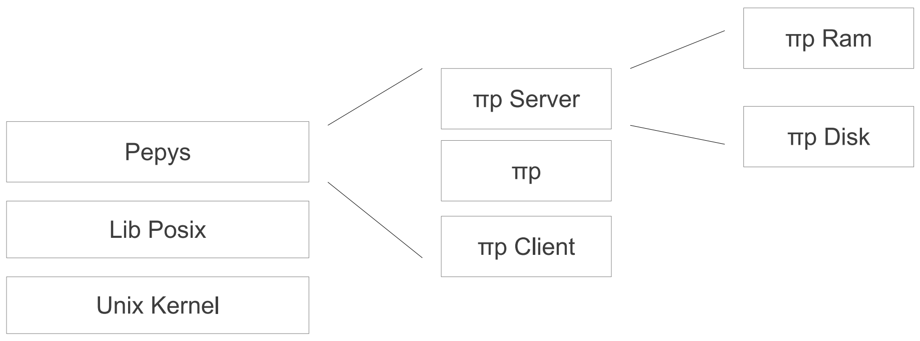

Figure 4.1: Porting on Linux implementation.

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 into the system, instead `nobody` to every entity present in the system (including both authenticated and not).

To understand better how these two entities are used, consider an ACL table. An ACL entry referring to the `others` entity applies to "every user logged and authenticated into 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 3.1).

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

4 Implementation Notes

4.1 Porting on Linux

The first step in our work was to unplug the Pepys file-system from its original structure (shown in figure 1.1) and therefore build a layer, called *Lib Posix*, in order to make the Pepys file-system runnable on UNIX machines.

In order to allow operations of swapping/loading objects from/into RAM, a new component has been added to the Pepys server, called *Pipdiskfs* (since the original structure provided only a RAM file-system).

Our porting relies on FIFO queues, provided by UNIX file-systems, in order to exchange ΠP messages between the server and the clients (however, we plan to switch to UDP-based communications).

The implementation is shown in Figure 4.1.

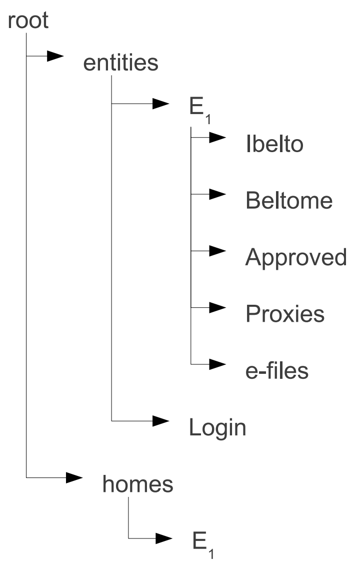

Figure 4.2: File-System structure.

The developed software included, in addition to the Pepys server, 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.

4.2 File-System Structure

Inside the file-system the entities are represented by special guards, which own a set of special files. Moreover, as we can see in Figure 4.2, each entity is associated with a `home` object (folder) over which it has full control.

Particularly each entity object guards (i.e., contains):

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 4.4.

4.3 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` and `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.

4.4 Delegation

As we said, two kinds of delegation are possible. In the on-line case, when an entity wants to delegate other entities it has to write the delegation in its `proxies` file. Specifically it has to indicate who is the delegee, its permission, an expiration date 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).

In the off-line delegation method, the delegee 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, is the delegation considered effective.

A valid delegation acts like a temporary ACL entry. However, the delegee might not have permissions to administer an object ACL, still be 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.

4.5 Authentication

When a client logs onto a Pepys server, it is not required to authenticate immediately, resulting in a session being in an unauthenticated state. 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, it has to write its (SHA-256) hashed password using a write command. The server compares the hashed password with the one stored in the entity password file, and, if they match, the entity session is upgraded to an authenticated state. From now on, the actual name 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 store 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.

Note that, 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.

5 Conclusions and Future Work

In this paper, an access-control model for the Pepys Internet-wide distributed file-system has been proposed, highlighting the main 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.

The paper 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: complete the implementation (under way) of digitally signed delegations; extend the delegations features including control of the delegation chain depth; integration of Pepys and particularly of the current authentication mechanism with properly designed cryptographic extensions to the ΠP protocol; evaluate the performance of the current ACM features, and possibly optimize the most recurrently used code paths.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] Stephen Tweedie. Ext3, journaling filesystem, 2000.

[14] 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.