Distributed file system over a multilevel secure architecture problems and solutions

Christel CALAS

CERT-ONERA
Département d'Etudes et de Recherches en Informatique
2 av. Edouard Belin
BP 4025
31055 Toulouse Cedex FRANCE
email: calas@tls-cs.cert.fr

Abstract. This paper presents the principles of a distributed and secure file system. It relies on M^2S machines and a secure network which control dependencies and avoid any storage and temporal covert channel. It describes how, from NFS (Network File System) principles, we adapt the organization and the structures to obtain practical services despite constraining controls performed by the hardware. Finally it proves that it is possible to obtain a practical distributed file system, with usable features without any compromise on security enforcement.

Keywords. Security, Distributed file system, Multilevel security, M^2S machine, secure LAN.

1 Introduction

Distributed environment is today an important feature to provide in a multiple host system. Particularly it is very efficient to offer distributed processing and remote access to information from any station connected to the system: many distributed systems and operating systems were built to that end. But well known techniques in classical environments become difficult to enforce when a high degree of security is required. Indeed distribution and security make a strange mixture where security, performances and access possibilities seem to be incompatible.

This paper aims at presenting a distributed file system that enforces the security of its data according to the multilevel security policy. It is based on using a particular machine called M^2S (*Machine for Multilevel Security*) [2] and a network assuring secure communication between hosts. The security of M^2S and of the network relies on the interpretation of the *causality* security model [3]. This model is based on controls of dependencies between objects maintained inside the system (*cell memory, driver ports,* ...). These controls enforce the security but entail constraints on the operating system, the file system and the applications running upon them. These constraints are not im-

posed to obtain security like *"You must do that or else security would be broken"* but rather *"Services are always secure and if you do not do that, your service could not run"*.

This enforcement of security requires an adaptation of usual functions and structures to obtain practical services. We illustrate this fact through a distributed file system example. As depicted in [2] or [6] security and distribution entail new problems both of security and functionalities and we explain how to resolve them and show that it could be realized in a simpler manner.

We first present hardware components and the local file system constructed on M^2S inside an adapted Unix operating system [1]. Then we depict the distributed file system principles and structure through a description of its actors and its organization. We discuss problems ensuing from this organization and propose solutions and finally we describe examples of the utilization of this MLS distributed file system.

2 Secure basis

2.1 M^2S: A Machine for Multilevel Security

M^2S is a secure machine built at CERT[1]-ONERA[2] which enforces multilevel security of its processes and its data avoiding any storage and timing covert channel. Its principles are depicted on Fig. 1. Its architecture is composed of two processors: one classical processor MC68020 and a MC68010 that is called the *security processor*. This security processor controls the dependencies between objects by controlling the accesses requested by the classical processor to memory cells and to device ports. The security processor plus some simple software constitute the *Security SubSystem* (SSS) of M^2S. A UNIX operating system has been developed over this machine and classical kernel has been modified to take account of the special functionalities of M^2S. It offers multilevel services as multilevel processing and multilevel file system through special calls and a multiplexed organization which are described more precisely in [1,2].

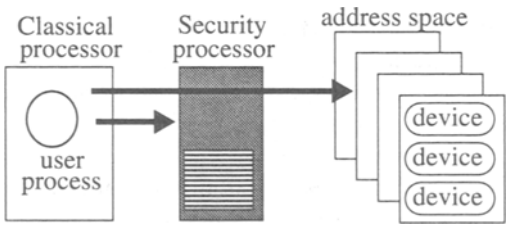

Fig. 1. M^2S security enforcement: Classical and security processors

1. Centre d'Etudes et de Recherches de Toulouse.
2. Office National d'Etudes et de Recherches Aéronautiques.

2.2 Secure LAN (Local Area Network)

Several M^2S machines can communicate through an Ethernet local network. The SSS has been extended in order to control the accesses of the classical processor to the communicator. As in case of local accesses, the SSS controls the dependencies between the processor, the bus and the buffer data. It allows the communications only when the communicators and the bus are at the same level. Moreover the SSS assures the timing and the level multiplexing of the bus in order to control the sharing of the network. This controls entails that:

- A *l* process may exchange information only with *l* processes.

3 Processing

Processes can run concurrently at different levels on M^2S machine. The SSS controls direct and indirect transmissions of information from high to low levels. Each machine is able to run processes up to a level corresponding to the maximum clearance assigned to the site. Processes are created either at login time by the login process or during sessions by user processes. Creating and killing processes can be done in accordance with the security rules due to the M^2S functioning:

- Processes at *l* level *(l > Unclassified)* may only create *l* children.
- Unclassified processes may create children at any *l* level managed by their host *(l ≥ Unclassified)*.

Having many hosts running multilevel processes and a secure LAN able to provide secure communications, multilevel distributed processing can be enforced.

4 Multilevel file system

Multilevel data are maintained on M^2S, inside multilevel, single-level files and directories. Multilevel files store data at different levels inside a single object [1]. The Unix kernel constructed upon M^2S provides the user with a multilevel tree-organized structure inside which files can be collected as depicted on Fig. 2. This structure and the associated services constitute the multilevel file system of a machine where the SSS controls the elementary flows. This fact necessitate to manage the underlying structures (*buffer, disk blocks, ...*) in order to be in accordance with these security controls. The following rules describe the perception of these controls at the user point of view:

- classified *l* files can be created only inside directories with same *l* level.
- classified *l* directories can be created both in unclassified or *l* directories.
- creation or deletion of a file or of a directory inside a given *l* directory can be made only by a process running at this l level.

Users handle the file system structure through classical Unix services (*open, read, write, close, cp, mv, ...*) and some new ones (*smkdir, sopen, ...*) that handle the multilevel objects.

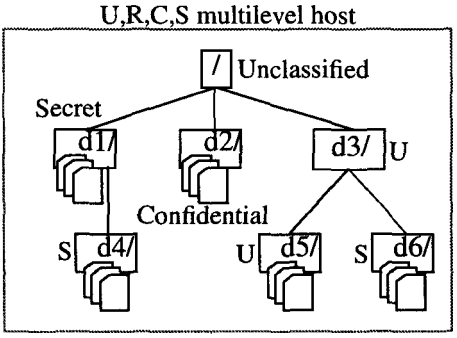

Fig. 2. Example of a multilevel tree in a multilevel file system.

5 Multilevel distributed file system

Consider a distributed environment composed of several sites and a communication channel. The sites maintain files and run processes for users. Every user can run its programs on any of these machines and so needs a common configuration that can be shared by these hosts. With respect to the file system, the distributed environment aims at providing the ability of accessing files of any sites from any machine.

But new problems appear in distributed systems: concurrent accesses, replication, cache coherency management due to remote location of resources and due to the entailed communications. Furthermore many others problems come from the controls enforced by the SSS. It is not problems of enforcement of security but rather problems of functionalities to provide in this secure and distributed environment.

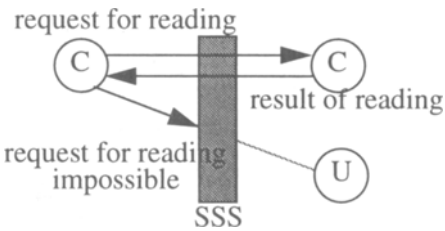

Fig. 3. Reading a remote file is forbidden by the SSS from high to low level

A classical problem occurs when a high level process attempts to read a low file maintained on a remote host at low level. It is a legal operation from a multilevel security point of view but it poses a serious problem in a distributed environment. Indeed in

a single-host system, read operations are exerted on passive objects whereas they are realized through exchange of messages in a distributed system. These exchanges occur between process requestor and a file manager process running on the remote site. We saw in 2.2 that the secure network allows only the exchanges at equal level. So it prevents the exchanges from high process to low process and the reading of a low remote file from a high process cannot be processed.

In [2] the authors present some solutions to resolve this problem. They suggest either to put the file manager in the SSS or to downgrade the read request or yet to construct multiple managers. Finally they chose the second solution for SDOS (*Secure Distributed Operating System*) consisting in downgrading read requests under the user control. On the contrary we chose to implement *multilevel managers*. It is a better solution for our point of view since, first, we need to maintain the SSS as light as possible avoiding to increase its size with unnecessary code and second, we prefer avoiding any downgrading mechanism which is a very constraining technique.

Knowing the controls enforced by the SSS we have chosen a structure and an implementation of a distributed file system which offer the bigger set of functionalities.

5.1 Overview

The system comprises at least one M^2S machine which offer multilevel processing and storage. It is characterized by a maximum level called *clearance* corresponding to the maximum level it can handle. A multilevel machine accepts logins at any level lower or equal than its maximum clearance so that a Secret multilevel site, for instance, is designed as a *[Unclassified, Restricted, Confidential, Secret]* machine. Let us remark that a multilevel machine can manage every level dominated by its clearance so it always manages the Unclassified level. This level is essential to construct multilevel services like multilevel managers. If it is necessary, several M^2S can be used in the distributed system.

The distributed environment contains also single-level sites characterized by their level *[level]*, providing single-level processing and storage. Single-level machines could be any kind of machines but they must be connected to the network through a TNIU (Trusted Network Interface Unit) controlling their accesses to the network. To obtain a correct functioning, this SSS must use the same protocol as M^2S.

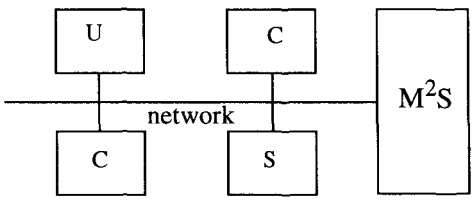

Fig. 4. Exemple of a hardware configuration of the distributed system

The last hardware component of the distributed environment is the secure network assuring the multilevel communications in a secure way. The goal of a multilevel file system is to offer some degree of sharing among the files stored on various machines in a practical manner. Now let us look at the principles of this sharing.

5.2 Principles

The sharing of files is based on a client/server paradigm. Servers handle locally files and offer services to remote clients for accessing them (*read, write, create, delete, ...*). Servers and clients may run both on single-level and multilevel sites so that there are single-level and multilevel servers. Multilevel servers run only on multilevel sites whereas single-level servers run both on multilevel and single-level machines. A multilevel server is composed of one process by levels. Each process runs the same piece of code. A multilevel server is characterized by the maximum level of processes which compose it and it contains a process for every level lower than this maximum level. A process of a l multilevel server manages files at l level but can also read files maintained by processes at level lower or equal than l and belonging to the same multilevel server. So these files are accessible from high clients through high server intermediary. Fig. 6 shows an example of Confidential multilevel server and Confidential single-level one.

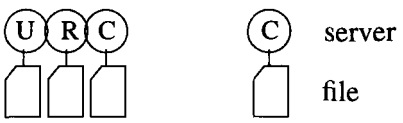

Fig. 5. Multilevel and single-level servers maintain files

Clients are always single processes at a given level running on a single-level or multilevel sites. They realize user processing and contact remote server when it is necessary.

Fig. 6. Clients

A client at *l* level can only converse with a process at l level and so operates on files with l classification but we saw that it can read also lower files through reading operations that can be performed by its equal server in the multilevel server. To sum up a client with *l* level can:

- access (*read, write*) *l* files of *l* single-level servers and of *l'* (l'≥l) multilevel servers.
- read files with level lower than *l* of multilevel servers.

From these rules we now present how clients contact servers and reference a file.

5.3 Mount-points

Accesses to remote files can be performed through direct addressing. When a user needs to realize an operation on a file he calls a special service and gives the name of the file and the address of the server maintaining it. This address must reference a process at the same level than the user or else the access will be rejected. This technique is simple but very constraining because it supposes than users know file location precisely at any time.

We prefer a more transparent technique based on the use of *mount-p*. Locally, clients keep mount-p tied up with remote files and perform access to these mount-p rather than to the files. A client does not know the real location of the file and only handles mount-p which can viewed as a pipe between this client and the file maintained by a server. A mount-p is one end of the pipe where clients see the remote file. Reading the pipe returns the file content and writing into the pipe modifies the file content. So pipe seems like the file for the user client.

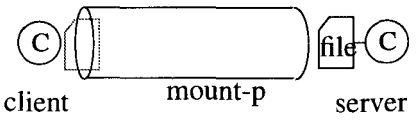

Fig. 7. Mount-p acts as a pipe pointed on a remote file

Mount-points are created either by clients themselves or by the kernel running on the client machine. As any other objects, mount-p have a level of classification. This level is defined by the classification of the process which creates them. It defines the set of operations that can be realized on them by a given process according to the security rules. The level of mount-p is equal to the level of the creating process and higher or equal than the file level.

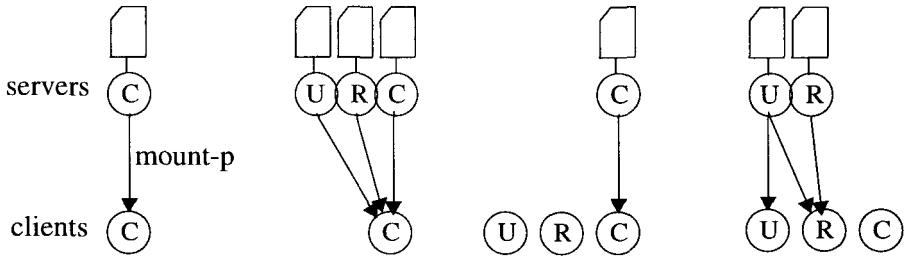

Fig. 8. Mount-pointsbetween single-level, multilevel, clients and servers hosts.

A mount-p pointing on a lower file is called an *inter-level mount-p*. An inter-level mount-p is an ambiguous object since it has a classification offering to realize some operations which cannot affect real file. One example is a C mount-p pointing to an U file. C processes can read, write, delete the mount-p but can only read the bound file. Fig. 8

depicts examples of every kind of mount-p that can be built with single-level and multilevel client and server sites. Mount-points are figured as arrows that point the client. An arrow between a client and a server figures all the mount-p between the client and files maintained by this server. Therefore they denote access possibilities and not only mount-p constructed effectively.

5.4 File system structure

We present now the implementation of the mount-p abstract structure in the distributed system. We chose a transparent naming scheme and a tree organization like the UNIX one. Every host maintains a local tree-structured file system inside which users (*or system*) can place mount-p. Mount-p creation is called *mount* operation. Mount-points do not reference files but directories and look like regular local directories. Their creation is done explicitly by indicating the remote host and the local name of the directory being bound (*see mount procedure*). On the other hand, we saw that usage is made transparently through the classical primitives and therefore a user can ignore the real kind of mount-p when it did not create the mount-p itself. Mount-p name is chosen by the creator and any string accepted by classical UNIX file system is allowed for this name.

In a tree-structured file system every directory but root, is stored in another one called its *parent*. There are rules which restrict mount-p creations and define the level of parent according to the remote directory level. These rules come from both security constraints and implementation choices.

- Mount-p created by a process with *l* level can only be inserted in *l* parent.
- Processes of a site with *ml* maximum level can mount mount-p only pointing on directories with parent lower than *ml*. Indeed the directories must be visible to be mounted.
- Mount-p on *l* directory can be created in any directory with level higher or equal than *l*.

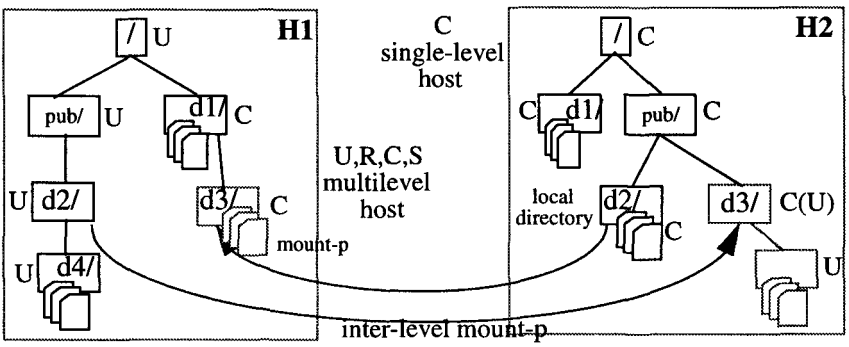

Fig. 9. Example of file system organization. Mount-points are stored as any other directories.

Fig. 9 presents an example of such a distributed file system containing a [U,R,C,S] multilevel host *H1* and a [C] single-level site *H2*. Each one contains local directories, files and mount-p to remote directories. On *H2* *"cd /pub/d3/d4"* is a valid command and can be followed by an execution of *"cat f1"* if *f1* is a file of */pub/d2/d4* on *H1*.

It shows also an example of inter-level mount-p through *d3/* on *H2* site.This mount-p has a Confidential classification but points on an Unclassified directory. Confidential is the real classification but operations are performed on the real Unclassified directory so that its behaviour (*result of operations*) always appears to *H2* processes as the behaviour of an Unclassified entity. Whatever the destruction or modification operations intended on this mount-p they concern only local structure on H2 and not real directory on H1.

Clients and servers constitute the heart of the system. They communicate through the secure network using an extension of TCP/IP and the protocol *SMAC* (*Secure Medium Access Control*) regulating operation requests and data exchanges.

5.5 Server

The role of servers is to maintain files, directories and to offer services in order to manage them from remote processes. There is only one server by host and it is the only way to offer directory access to a remote site. Therefore files of a serverless host could not be accessed remotely.

Servers receive requests from clients describing the directory operations requested (*change directory, file creation, file destruction, file transfer*) then realize the service and send the response. They use the local file system of the host on which they rely to provide the realization of the operations. They perform the communication with clients and control the creation of mount-p established on one of their own directories. These controls prevent a mount operation on an inexistent directory and avoid so unnecessary mount-p creation on client side.

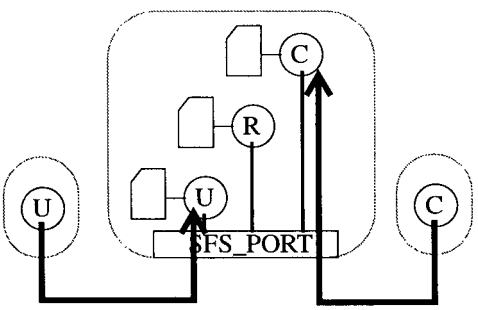

Fig. 10. Port numbers are multiplexed by level and clients reach only processes at their level

Servers could be single-level or multilevel according to type of their host and the kind of clients they must be able to serve. In the distributed system presented here, there

is only one multilevel server which run on the M^2S machine. Clients at l level contact servers at l level. The multilevel server is composed of processes running at different levels and waiting connections on the same TCP port number. This port is called *SFS_PORT* and it is reserved in every machine for the file system server. So from the address of a machine, a process is able to reach its file system server by connecting to *address:SFS_PORT.*

Port numbers are multiplexed by level so that connections to port *SFS_PORT* wake up the server having the same level as the requestor. The multilevel server is initialized from Unclassified level which is always processed in multilevel machine. Its creation is made by Unclassified process which creates the other classified processes composing the server. This initialization is made at boot time in the *initfs()* procedure. Every process of the multilevel server binds itself to the *SFS_PORT* port number and waits for client requests on this port. Monolevel servers are identical to multilevel ones except that they are single process.

Servers of the file system are stateless as NFS servers so that they do not keep any information on client mount-p, connections and requests. Therefore clients must indicate the whole information necessary to perform an operation inside every request. The advantage of stateless servers is that they can crash and later rejoin the system without any drawbacks than the lack of service during their malfunction. So global state is conserved and does not need to be restored.

Client requests have all the same structure presented on Fig. 11. They contain the identification of the operation requested *(opid)*, the local path *(local path)* of the file or the directory on which the operation must be intended by the server and finally the length and the data necessary to provide the operation (*name file in file creation, offset in the file and so on*). Every operation corresponds exactly to one local service and *localpath* is directly used by the server to perform the operation through call of the local file system. Servers receive the result of the local execution and send it to the client. Error report is generated when the operation fails.

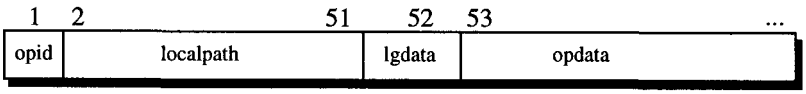

Fig. 11. Structure of requests provided by a client to a server

Servers are managed only through one primitive called *initfs()* which is executed at boot time (at Unclassified level in the multilevel machine) and realizes server creation. Server characteristics are defined in the local file *"/SFS.INIT"* which describes if a server must be created on this host, its type (*multilevel or single-level*) and its level. Of course these informations are restricted by the abilities of the site to manage the levels. For instance a Secret multilevel server becomes a Confidential single-level server on a

Confidential single-level site and on the other hand, single-level server may be created on a multilevel site in order to restrict the remote accesses to one given level.

5.6 Client

Clients are the second components of the multilevel distributed file system. A user process intending an operation on a mount-p becomes a client of the corresponding server. Ideally, this transformation is transparent to the user but in fact it could be perceptible through the difference in response time between a local and a remote response due to the communication duration. Cache mechanisms can be used to improve the response time in the better case where file content is in the local buffer. [4] presents the security problem due to the cache utilization and we discuss it in section 5.7.

Mount-points are the only means to reach remote files and directories. They maintain the correspondence between a virtual local directory (*the mount-p*) and the real remote directory attached to it. This correspondence is only maintained on a client site. In the following text we will use the word mount-p instead of the corresponding virtual local directory. From transparency principle, user processes use mount-p as any other local directory both in commands and file system calls. References to these mount-p are then transformed on remote communications with the server handling the real directory.

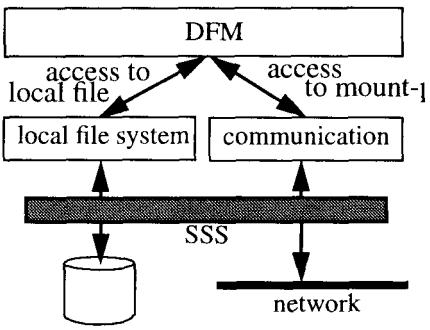

Fig. 12. The Distributed File system Manager switches user accesses either to local file system or to a server

These conversions are performed by the *DFM* (*Distributed File Manager*) which is a special module located over the local file system. Any call to file system reaches the DFM which switches it either to local file system or to communication layer. DFM uses an extension of classical inodes to determine the type of directory referenced and find its real address. Precisely extensions concern *inode mode* field which can take the new value *MODE_REMOTE* indicating a mount-p to a remote directory. The *r_localpath* field receives the localpath of the mounted directory (*real name on server host*), *r_level* keeps the level of this directory and *r_server_level* contains the type of server (*single-level or multilevel*) and its level. The two last fields are used to anticipate errors returned by the SSS when a process intents a forbidden access. Remember that these fields are

not used in any way to enforce security but only to gain time in avoiding requests which will be anyway not accepted by the SSS.

```
struct inode_DFM
        {
        Struct inode inode_l;
        char        r_localpath[LG_LOCAL_PATH];
        int         r_level;
        int         r_server_level;
        }
```

For example at current level *current_level* it is not necessary to intent an access on an host at level *r_server_level* lower than *current_level (current_level > r_server_level).* As we saw in 5.4, mount-p on single-level host have not necessarily the same level as the remote directory and *r_level* is used to avoid request impossible to realize on the real directory (*destruction or modification for example*). *r_level* is also used by DFM to display correct information in *ls* command for example where level of a mount-p takes *r_level* value and not the real level of the mount-p. So users have all the information to understand the result of their operations (*modification rejection for example*).

Consider now the DFM primitives *mount, umount, lmount* and *initfs* managing the mount-p. For every one we present its syntax and a description of its functionalities. The syntax is the same as classical UNIX description where [...] denotes an optional parameter and <...> contains the description of a real parameter.

- mount [-f] <machine>:<remote directory> <mount-pname>

mount creates a new mount-p, in place of the existent local directory named <*mount-p name*>, which points on <*remote directory*> of the <*machine*> site. From this creation, the <remote directory> content (*files and directories*) is transparently present in the directory <mount-p name> of the local host and can be manipulated as any other directories. *Mount* operations entail a communication with the corresponding server which verifies the existence of its local directory and sends an acknowledgment or an error both when it does not exist and when none server can be reached at current level. The server sends the new location of the directory afetr a file system reorganisation. Optional parameter -f forces the mount-p creation even when an error occurs. It is usable when the corresponding server has not yet be initialized. Mount-p creation performs a directory modification so that DFM allows creation (*due to SSS controls*) only when it takes place in a directory having same level than the process intending it. Therefore mount-p take the level of their creator and never the level of their bound remote directory.

- umount <mount-p name>

umount deletes a mount-p. It entails a modification of parent directory and must be executed at parent level. Only the mount-p is concerned by this command and in any way the remote directory bound with it.

- lmount

lists the whole mount-p mounted on the host with real directory names and levels associated. It is useful for administrator users intending debug procedures.

- initfs [-g]

We saw this primitive in 5.5 for server initialization but it concerns also the client part to initialize the initial tree-structure at boot time. It reads a setup multilevel file called *"/TFS.INIT"* containing the description (*mount-p name - remote directories*) of the initial structure, at every level managed by the host. On the multilevel site an Unclassified boot process creates a process for every level managed by the host. Each one of these processes realizes the mount operations at its level and then dies. Option *-g* executes a global initialization. In global initialization the system uses a remote *"/TFS.INIT"* called *global file*, rather than the local *"/TFS.INIT"*. This global file is stored on a host called *global host* (classically the M^2S machine) whose identification is given in *GLOBAL_SERVER* variable. Each initialization process contacts the process of that global server and requests the content of the setup file. Then it realizes the mount operations and dies. Of course multilevel server must be present on *GLOBAL_SERVER* host or else local file will be used. *Initfs* execution must be integrated in the boot procedure.

5.7 Problem discussion

Transparency is a requirement that is difficult to meet in a distributed and secure environment because of the set of complex actions executed inside a simple operation. For example reading a file implies a connection to the server and exchange of part of its content. It takes a not inconsiderable time to execute these sub operations and then reading a file becomes a slow service. Therefore caches can be maintained on the client hosts.

But in [2, 5] authors explain that there is a possibility of timing covert channels in using these caches since they modulate the time of file access. Assume that a high process is reading a low file for the first time on a machine. This operation creates a cache on the machine. Then a low process intents a read on the same low file. The response time depends on the cache existence an so on high process actions. So there is a timing covert channel.

In the system presented in this paper, this timing covert channel is inexistent because either cache is a low memory and so high process cannot place the file content in it or it is a high buffer and low process cannot read it. In both cases the same cache may not be used by processes running at different level.

Fig. 13 shows a functionality problem specific to a multilevel system: the problem of Tantalus mount-p. Tantalus mount-p are directories visible to high processes but inaccessible for them.

This problem appears when a low process on the multilevel host creates a mount-p to a low directory maintained in a low single-level site. The example shows an Unclassified mount-p which points a directory stored on an Unclassified single-level site *H2*. Being an Unclassified mount-p, it is visible for high processes. U clients of *H1* contact [U] server of *H2* to realize operations on the directory. Any other high clients of *H1* cannot contact this server and so cannot access to the remote directory whereas they see the Unclassified mount-p.

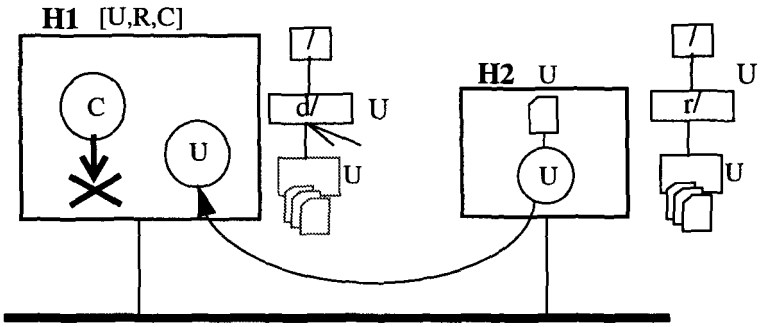

Fig. 13. Tantalus mount-p: processes see them but cannot reach.

It is the Tantalus mount-p problem. We have at least two solutions to solve this problem. Either to reject all creations of mount-p on a site whose levels are not managed by the server or to allow this problem but offering a manner to console frustrated processes. We chose the latter solution which is based on the multiplexing of inodes in memory.

When a process realizes a directory operation, the DFM loads in memory the inode of the directory. This inode is stored in memory inside the *in-core inode table*. This table is multiplexed by levels and the inode is stored in the table classified at the process level. In a mount operation, the information about the remote directory pointed by this mount-p are stored in this table (see structure of inode_DFM in 5.6). Therefore processes classified at various level can create mount-p with the same name (in place of the same directory) but which point various remote directories. These mount-p are called *multiplexed mount-p*.

Fig. 14 depicts its principles based on inodes table multiplexed by levels in memory and on a searching procedure *namei()* running at current level. When a process intends to access to a directory *(mount-p) d/* it calls a file system service and furnishes the name of this directory. *Namei()* is then executed to find the inode associated with the given name. This function realizes its research in the in-core inode table classified at the current level and so reaches the directory stored at this current level.

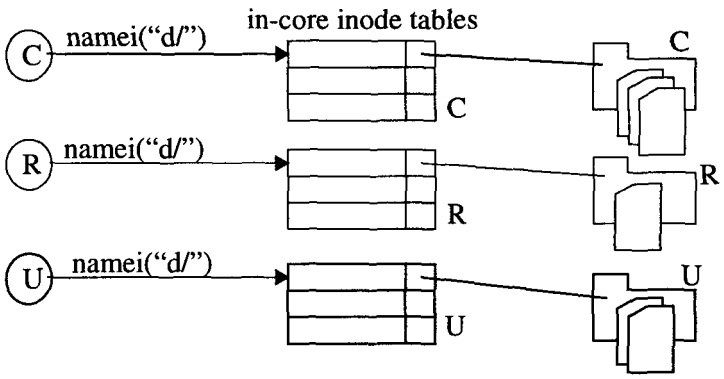

Fig. 14. Principles of mount-p multiplexing

The advantages of this functionning is that low mount-p will be visible only to low processes and not to high ones (and conversely) hiding the Tantalus mount-p problem. Low process can also perform a copy of this file from the single level site to its multi-level host. Rather than to impose a rigid solution, the freedom is given to users to organize clients, servers and mount-p in order to avoid such a problem.

The multiplexed mount-p can also be used to assure a multiplexed service. Fig. 15 shows an example of such a utilization. In this example, U and C users need to run a program which displays the list of books classified at their level. Each list is maintained by a server and the program accesses to them in the same directory (*mount-p*) *"BOOKS"* whatever the level. Transparency is so given between level and multiplexed service is assured.

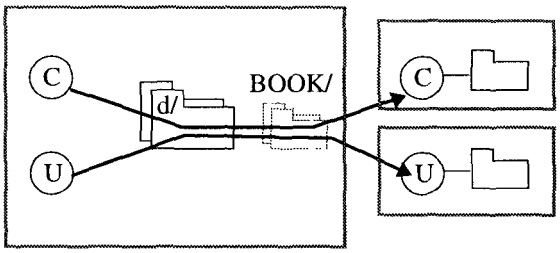

Fig. 15. Mount-pmultiplexing can provide transparent multiplexed services

6 Sharing

The following text describes the real possibilities of sharing provided by this file system. If we consider an environment composed only of single-level sites (*at identical levels*) this system offers the same functionalities as the *NFS* protocol. But advantages

came in an heterogeneous system where users, files and sites run at different levels. In this environment mount-p usage offers the sharing of information between various levels. The classical sharing technique is based on the use of the multilevel host M^2S maintaining shared files and of clients running on single-level hosts [2]. Low processes modify file content inside the multilevel site and high processes read it when they need.

The implementation of this sharing technique in the distributed system presented here, relies on the usage of the multilevel server and of mount-p created on the client hosts (*low and high*) and pointing on the shared file. Remark that this organization provide the sharing of information even if client hosts are single-level. So it is possible to construct a multilevel distributed system with only one multilevel host (*the server*) and several single-level sites and so reduce its cost and its complexity.

Multiplexed mount-p are a new functionality offered by this system. On one hand there are several single-level hosts running servers and on the other hand users on the multilevel site which access transparently to the site running at their level. This construction is very important to assure the perfect transparency necessary to run a program from any level. The multiplexing technique can be used either on the data files or directly on the programs.

In any case, the users (*or an organizer*) are in charge of constructing the system and of placing the files in order to offer the sharing of data in a practical way. Indeed if a user decides to maintain a file inside a single-level site, the system will be never able to offer accesses from another level. But, on the other hand, it offers the possibilities of publishing a file by copying it in a mount-p pointed on a server. The file will be so placed on the server and will be accessible by any client handling a mount-p on this server. Therefore and only if they decide it, the users are the possibility to allow access on their files. But whatever the organization chosen by the users, the security is always assured.

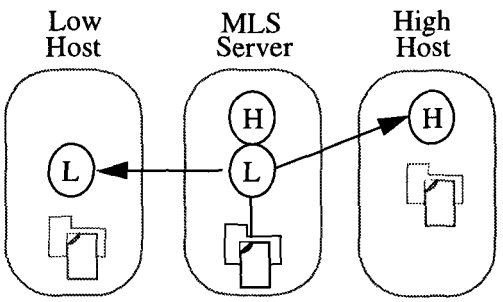

Fig. 16. Example of MLS server

7 Summary

Distribution functionalities and security are often difficult to mix. They seem to be antagonistic and pose problems to any designer of distributed file system. Generally compromises must be made and some lacks of security are commonly accepted. In this project compromises are made on functionalities and not on security. Indeed, this project aims at assuring the functionality of a practical distributed file system over secure basis and not the security of a distributed file system. We showed that despite the controls enforced by the hardware, it is possible to construct a practical system offering the classical services and many new ones (*multiplexed views*). Of course it entails some problems but we saw that solutions could always be constructed and that they are quite simple.

Further, we would discuss about other kind of security as discretionary policies. Client/server is a practical paradigm to realize controls on user accesses since the servers handle the whole accesses intended on one of their files. It would be so the main actor of this security. It would be also interesting to study how we could use process migration to realize accesses impossible directly or yet redundant servers.

8 References

1. B. d'Ausbourg, C. Calas
   "Unix services for multilevel storage and communications over a LAN"- Proceeding of the Winter 93 USENIX Technical Conference, San Diego, 1993
2. B. d'Ausbourg, J-H. Llareus
   "M2S: A Machine for Multilevel Security"- Proceeding of Esorics'92, Toulouse, November 23-25, 1992
3. P. Bieber, F. Cuppens
   "A Logical view of Secure Dependencies"- Journal of Computer Security, Vol 1,Nr 1, 1992
4. Thomas A. Casey Jr., Stephen T. Vinter, D.G. Weber,
   R. Varadarajan,D. Rosenthal
   "A Secure Distributed Operating System"- Proceeding of IEEE Symposium on Security and Privacy, Oakland, April 18-21, 1988.
5. Glenn H. MacEwen, Bruce Burwell, Zhuo-Jun Lu
   "Multi-Level Security Based on Physical Distribution"- Proceeding of IEEE Symposium on Security and Privacy, Oakland, 1984.
6. Richard E. Smith
   "MLS File Service for Network Data Sharing"- Proceeding of Computer Security Applications Conference, Orlando December 6-10, 1993