
Implémentation des droits de lecture dans PASTIS

LIP6

Octobre - Décembre 2006

Julien Bachmann

e-mail: [julien.bachmann \[at sign\] gmail.com](mailto:julien.bachmann[at]sign@gmail.com)

Table des matières

1	Introduction	2
1.1	<i>PASTIS</i>	2
1.2	Contexte	3
2	Travail effectué	6
2.1	Axes de recherche	6
2.1.1	L'encryption convergente	6
2.1.2	Utilisation des ACL	8
2.1.3	Encryption asymétrique	14
2.1.4	Etude du système de fichiers du LIP6	15
2.1.5	Solution choisie	17
2.1.6	Tableau de comparaison entre EKB et ACL	17
2.2	Implémentation	18
2.2.1	Ajout du bloc USER	18
2.2.2	Modification du gestionnaire de sécurité	19
2.2.3	Mise en place dans le code existant	19
2.2.4	Modifications pour intégrer les ACLs	22
2.2.5	Déroulement du projet	23
2.3	Tests	24
2.4	Mesures de performances	26
2.4.1	Ecriture et lecture de blocs	26
2.4.2	Comparaison avec d'autres systèmes de fichiers	26
3	Conclusion	28

Chapitre 1

Introduction

1.1 *PASTIS*

PASTIS est un système de fichiers pair-à-pair en lecture-écriture d'architecture entièrement répartie. Tout comme un système de fichiers UNIX, *PASTIS* utilise une structure arborescente pour représenter les données, à l'aide d'inodes et de blocs de données.

Afin de créer un système pair-à-pair, *PASTIS* stocke ses blocs dans une table de hachage répartie (appelée *DHT*, *Distributed Hash Table*) - *Past*, qui elle repose sur le réseau pair-à-pair *Pastry*.

L'utilisation de ces deux composants permet d'avoir un système tolérant aux fautes et hautement disponible.

Afin d'être utilisé sur un système, l'implémentation, écrite en Java, utilise les interfaces du système de fichiers NFS et FUSE qui permettent d'avoir des systèmes de fichiers dans le mode utilisateur.

Pour le moment, uniquement le contrôle d'accès en écriture a été implémenté dans *PASTIS*.

Étant donné que ce système de fichiers est réparti, on ne peut pas mettre en place un système de protection comme celui qui existe sur les systèmes de type UNIX. En effet, vu que les fichiers se trouvent sur un réseau de noeuds, si l'un de ces noeuds est en fait un attaquant (noeud dit "byzantine", c'est à dire qui ne respecte pas les spécifications soit à cause d'un bug logiciel, soit parce qu'il est contrôlé par un utilisateur malveillant) il peut ne pas tenir compte du champ de bits qui correspond aux droits et ainsi lire le fichier alors qu'il n'avait pas les droits suffisants. Au contraire, dans un système UNIX, le verrouillage est réalisé au niveau du système et on ne peut ainsi pas forcer l'accès à un fichier si nous n'en sommes pas propriétaire ou le super-utilisateur (*root*).

La seule solution pour pallier ce problème est d'utiliser la cryptographie afin d'encoder les données qui se trouvent sur le système de fichiers. Ainsi, seules les personnes ayant la clef pourront décrypter les données.

Une des particularités de *PASTIS* est qu'il n'y a pas de noeud central. On ne peut donc pas envisager un système de type PKI (*Public Key Infrastructure*) afin de gérer la distribution des clefs. Dans un tel système, on se réfère à un serveur central qui distribue les clefs.

De plus, dans un système de fichiers, les performances sont un facteur assez important car l'utilisateur ne doit pas être pénalisé par le fait que ce système soit réparti, mais plutôt en tirer partie. Il faut donc que la solution au problème respecte des contraintes en matière de taille de données ainsi qu'en utilisation de la bande passante.

Le but de mon travail était donc de trouver une solution afin de mettre en place un contrôle d'accès en lecture par utilisation de la cryptographie, puis de l'implémenter dans le modèle de recherche qui est codé en Java.

1.2 Contexte

Cette partie va présenter le fonctionnement de *PASTIS*, ainsi que ce qui est déjà en place, ce qui permettra de mieux aborder le reste du document.

Comme indiqué précédemment, *PASTIS* utilise principalement deux types de blocs logiques, les inodes et les blocs de données. Eux mêmes vont être représentés par d'autres types de blocs lorsqu'il va s'agir de les stocker dans la DHT de *Past*.

Ces deux types de blocs sont les CHB (*Content Hash Block*) et les ACB (*Access Control list Block*). Ces derniers ayant été ajoutés pour contrôler l'intégrité des blocs et implémenter les mécanismes de contrôle d'accès. L'utilisation d'un type de bloc ou de l'autre va dépendre de l'utilisation que l'on veut en faire.

Les CHB sont utilisés pour stocker des données qui ne sont pas modifiables. La clef de stockage dans la DHT va être le hash du contenu du bloc. En cas de modification du contenu du bloc, nous allons devoir changer toutes les références vers le CHB modifié afin qu'elles utilisent la nouvelle clef de stockage.

Les ACB, quand à eux, ont un couple de clefs privées/publiques qui est associé à leur création. La clef de stockage est le hash de la clef publique.

En cas de modification, nous n'allons pas avoir création d'un nouveau ACB. C'est pour cela que chaque ACB contient un numéro de version qui va nous servir à identifier la dernière version et ainsi éviter des attaques de type *rollback* (consiste en la réinjection de blocs correctement signés mais plus anciens que la version courante).

Les ACB permettent aussi de gérer le contrôle d'accès en écriture, car ils utilisent un bloc logique ACL (*Access Control List*). Sur les systèmes de type UNIX, les ACLs permettent d'avoir une gestion des droits plus fine. On ne va plus, juste donner les droits en lecture à un groupe de personnes, mais à une liste de personnes qui peut être de groupes différents.

Les ACL ont tout comme les CHB, une clef de stockage qui correspond au hash du contenu du bloc. Dans la solution actuelle qui gère les droits d'accès en écriture, cette particularité permet d'éviter à un attaquant de modifier une ACL.

En effet, si un attaquant modifie un ACL, il va changer la clef de stockage. Or cette

dernière est contenue dans une partie signée par le propriétaire du fichier dans l'inode.

Le figure 1.1 représente la gestion actuelle des droits en écriture dans *PASTIS*.

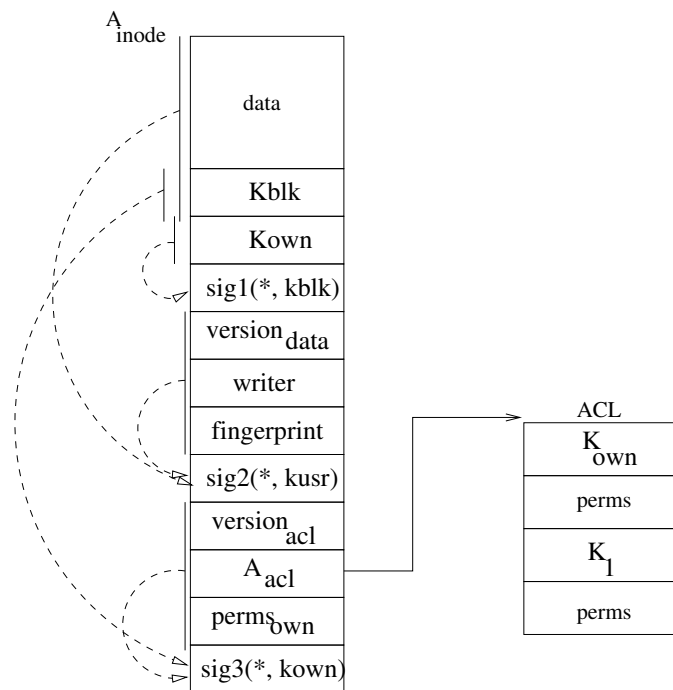


FIG. 1.1 – ACB et son ACL

Chapitre 2

Travail effectué

2.1 Axes de recherche

Dans cette section, nous allons voir les différentes solutions proposées afin de mettre en place les droits en lecture dans *PASTIS*.

2.1.1 L'encryption convergente

L'encryption convergente est la première solution vers laquelle je me suis tourné. Cette technique a été inventée par *Microsoft Research* pour être utilisée par leur système de fichiers *FarSite*. Le but de cette méthode est de fusionner les blocs de données similaires afin de gagner de la place.

La méthode d'encryption est alors la suivante :

- * on calcule un hash cryptographiquement fort (cf. annexe B.3) du contenu du bloc (ex. MD5)
- * on encrypte les données du bloc en utilisant une méthode d'encryption symétrique (ex. AES)
- * on encrypte le hash avec la clef publique de chaque personne autorisée à lire le fichier
- * on stocke les hash encryptés sous la forme de méta-données reliées au bloc (ex. RSA)

Afin de stocker ces méta-données, on peut les rajouter au CHB qui correspond au bloc de données. La clef, étant déduite du contenu du bloc, n'est elle reliée qu'à un seul bloc. On obtient donc un bloc de données comme celui-ci de la figure 2.1 (K représentant une clef publique et k une clef privée).

La lecture des données se fait, pour un utilisateur U de la manière suivante :

- * il récupère le bloc A
- * il vérifie que la clef de stockage est bien la bonne
 $A == \text{SHA-1}(\text{data} + \text{salt})$
- * il cherche sa position dans la méta-donnée en utilisant le hash de sa clef publique
- * il décrypte le hash en utilisant sa clef privée
- * il décrypte les données en utilisant le hash

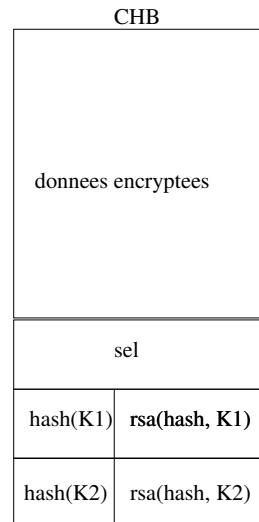


FIG. 2.1 – bloc de données avec l’encryption convergente

Ajout et suppression de lecteurs

L’ajout d’un nouveau lecteur se fait très simplement : le propriétaire du fichier n’a qu’à parcourir les différents blocs du fichier duquel il veut changer les droits afin de rajouter le hash encrypté pour le nouveau lecteur.

Pour la suppression d’un lecteur, on pourrait envisager un premier cas. Supprimer le lecteur des méta-données uniquement lors du prochain changement dans les données du bloc (cela aura pour incidence de créer un nouveau hash), car on ne peut pas l’obliger à oublier ce qu’il a déjà lu (il aurait très bien pu le copier). Cette méthode a l’avantage de ne pas faire d’opérations inutiles.

Cette méthode a néanmoins un défaut. Elle permet à une personne dont les droits ont été retirés de voir l’avancement du fichier. Elle peut voir quels vont être les blocs modifiés, ou non, en fonction de ceux qui n’ont pas été réencryptés.

C’est pour cela qu’il vaut mieux utiliser une autre méthode qui consiste à encrypter tous les blocs du fichier lors de la suppression d’un lecteur.

Avantages

Un des avantages de l’encryption convergente est le gain de place sur le système de fichiers résultant de la fusion des blocs encryptés similaires.

Lors de tests, *Microsoft Research* a montré que l’on pouvait gagner jusqu’à 30% de place sur le système de fichier. Ce résultat est néanmoins à nuancer, d’après les articles disponibles sur le site du projet *FarSite*, il semblerait que lors des tests, les binaires contenus dans *Program Files* ont été pris en compte.

Un autre avantage, est la facilité de changement de droits grâce à la modification des méta-données.

Inconvénients

Le stockage des méta-données par bloc et non par fichier est assez problématique, car il représente un volume non négligeable de données. Ce volume n'étant pas rattrapé par le fusionnement des blocs de données lorsque le taux de similitude des blocs passe en dessous de 15% (chiffre plus représentatif que celui de *Microsoft* qui lui tient compte des binaires).

2.1.2 Utilisation des ACL

Les ACL (Access Control List) ont été mis en place dans *PASTIS* lors de la mise en oeuvre du contrôle d'accès en écriture. Sur un système de type UNIX, elles servent aussi bien à gérer l'écriture que la lecture, il est donc normal de penser à une solution qui les utilisent dans notre cas.

Dans cette méthode, la clef d'encryption va être générée de façon aléatoire. On reprend le principe de stocker la clef sous une forme encryptée à l'aide de la clef publique de chaque utilisateur. Mais cette fois-ci, ce ne sont pas les blocs que nous allons fusionner, mais les ACL équivalentes.

Une modification va aussi être effectuée au niveau du bloc de type USER, afin de lui rajouter un champ. Celui-ci va permettre de contenir la clef d'encryption, elle même sous forme encryptée, des fichiers pour lesquels l'utilisateur est le seul à avoir un accès en lecture et en écriture. Toutefois, les écrivains ayant besoin d'encrypter les données lors de la sauvegarde d'une modification, sont comptés comme lecteurs.

Nous allons donc avoir un bloc USER représenté comme dans la figure 2.2 (les autres informations comme le nom et l'e-mail ne sont pas encore clairement définis).

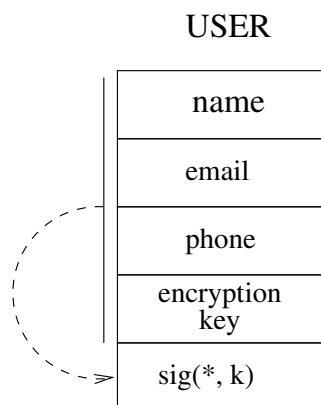


FIG. 2.2 – bloc USER

Création d'un fichier

Dans cet exemple, nous partons du principe que lorsqu'on crée un fichier, les droits par défaut correspondent à des droits en lecture et écriture pour le propriétaire uniquement.

En partant de cette hypothèse, nous allons avoir beaucoup de fichiers avec ces droits. C'est pour cela que nous allons utiliser la clef stockée dans le bloc USER afin d'encrypter les données de ces fichiers.

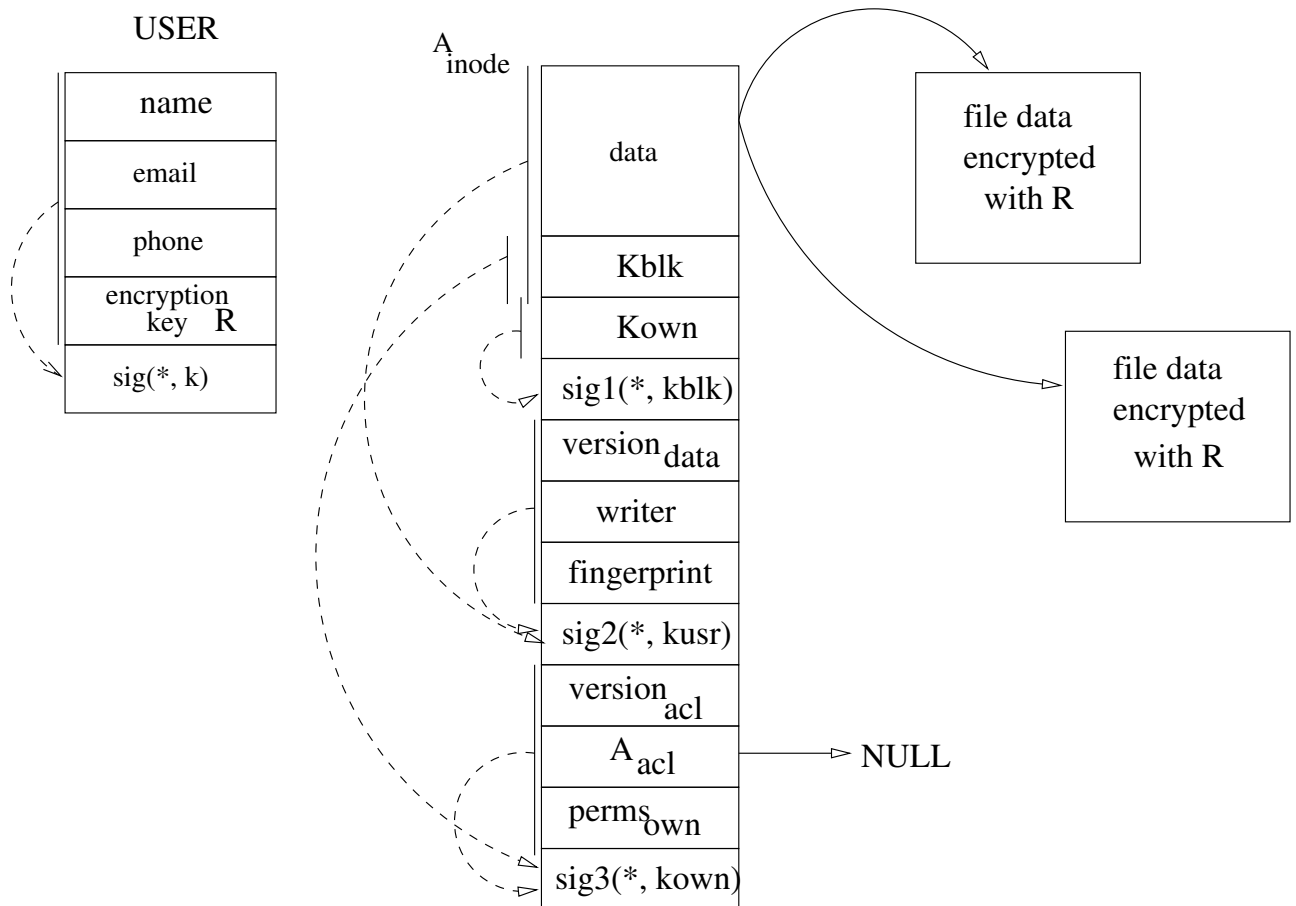


FIG. 2.3 – création d'un fichier

De ce fait, nous n'allons pas avoir de bloc ACL relié à l'inode, car toutes les informations relatives aux droits sont dans le bloc USER (clef d'encryption) et dans l'inode (owner_perms). Cela va nous permettre de réduire la place prise par les informations sur les droits dans le système.

Nous allons donc nous retrouver dans un cas comme celui de la figure 2.3.

Dans cette configuration, la seule chose que pourrait faire un noeud byzantin, serait de modifier une des répliques du bloc USER afin de le rendre inutilisable. Mais dans ce cas, nous n'avons qu'à aller chercher une autre réplique dans la DHT.

Partage d'un fichier avec un autre utilisateur

Afin de partager un fichier avec un autre utilisateur, nous n'allons plus utiliser notre clef d'encryption qui se trouve dans le bloc USER, mais une clef qui va être utilisée pour tous les fichiers que nous allons partager avec cet utilisateur.

Le principe est le suivant :

- * on va simuler l'ACL résultant de l'ajout d'un nouveau lecteur pour le fichier.
- * on utilise le hash de cet ACL afin de vérifier s'il n'existe pas déjà.
- * dans le cas où il n'existe pas, nous allons générer une nouvelle clef afin d'encrypter les données du fichier. Puis nous allons mettre cette clef dans le nouvel ACL qui va être à présent utilisé pour le fichier.

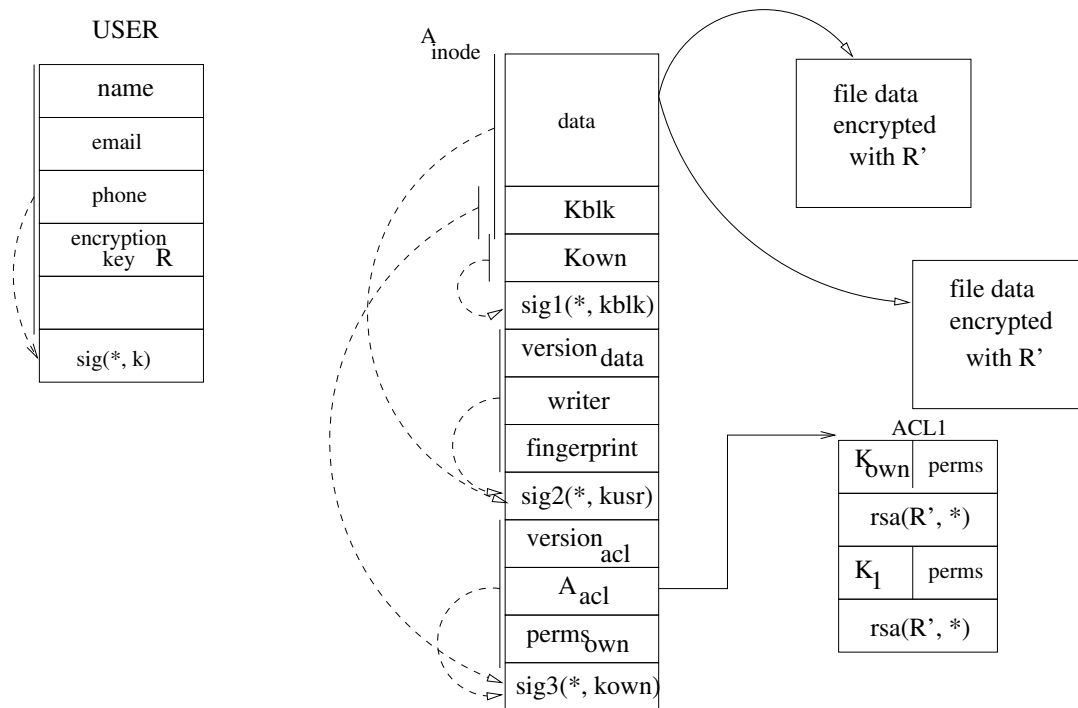


FIG. 2.4 – partage avec un autre utilisateur

- * dans le cas où il existe déjà, nous allons prendre l'adresse de l'ACL existant afin de la mettre dans l'inode. Puis nous allons réencrypter les données du fichier avec la clef du nouvel ACL.

Ce qui donne la figure 2.4

Cela semble fonctionner, mais nous avons tout de même un problème.

Problème 1 : La clef d'encryption qui, se trouve dans l'ACL est encryptée à l'aide de notre clef publique, un noeud byzantin pourrait tout à fait générer de faux ACLs qui nous contiennent afin que nous les utilisions pour encrypter nos fichiers. De cette façon, il a gagné le droit de lecture sur les fichiers utilisant cet ACL car il a lui même généré la clef.

Solution 1 : Une solution à ce problème serait alors de signer un ACL avec la clef privée du propriétaire du fichier, afin de certifier que c'est bien lui qui a créé l'ACL et non pas un attaquant (figure 2.5).

Problème 2 : Malheureusement cette solution a un inconvénient majeur : afin de vérifier que c'est bien le propriétaire de l'un des fichiers qui utilise un ACL qui a signé ce dernier, nous allons devoir parcourir le système de fichiers à la recherche des fichiers utilisant cet ACL afin de trouver un fichier qui appartient à la personne qui a signé l'ACL.

Parcourir entièrement le système de fichiers est, bien sûr, exclu (utilisation de trop de bande passante), une solution pourrait être que l'on puisse savoir pour chaque utilisateur quels sont les ACLs qu'il a créé.

Solution 2 : Pour faire cela, nous ne pouvons pas utiliser un point central, car le système est entièrement réparti. Nous allons donc ajouter cette information dans le bloc USER, soit en l'intégrant au bloc, soit par l'intermédiaire d'un nouveau bloc dont l'adresse serait contenue dans le bloc USER.

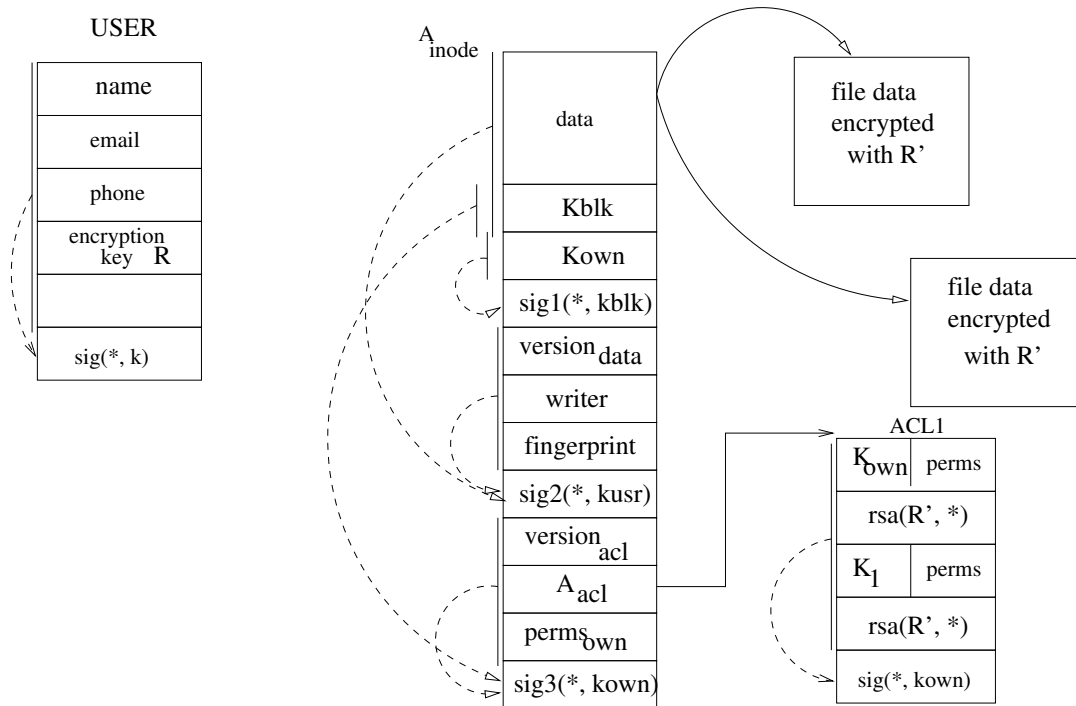


FIG. 2.5 – ACL signé par propriétaire

C'est dans l'optique de la seconde solution que je propose un nouveau bloc, le *CAB* (*Created ACL Block*).

Notre solution ressemble maintenant à la figure 2.6.

Problème 3 : La vérification de l'ACL va être effectuée à chaque lecture et écriture du fichier. Cette solution est donc correcte, mais elle demande trop de ressources. Nous devons donc la laisser de côté et chercher une méthode plus simple afin de résoudre notre problème.

Solution 3 : Depuis la mise en oeuvre du contrôle d'accès en écriture, la clef de stockage d'un ACL est le hash des clefs publiques et des permissions, le tout par ordre croissant de clefs publiques. Nous devrions donc garder le même système, afin de pouvoir trouver facilement un ACL similaire.

Nous allons donc rajouter une signature sur les encryptions de la dite clef à la fin du bloc ACL. Cette signature va être faite avec la clef privée du propriétaire du fichier. Les ACLs ne se partagent donc plus qu'entre les fichiers d'un même propriétaire.

Ce qui va nous donner la figure 2.7

La vérification va donc s'effectuer de la manière suivante :

* $A == \text{SHA-1}(\text{Kown} + \text{perms_own} + \text{K1} + \text{perms_1})$

* $\text{verify}(\text{rsa}(\text{R}, \text{Kown}) + \text{rsa}(\text{R}, \text{K1}), \text{sig})$

Dans le cas où plusieurs fichiers se partagent les mêmes droits, nous allons avoir le même ACL, ce qui va se traduire par la figure 2.8

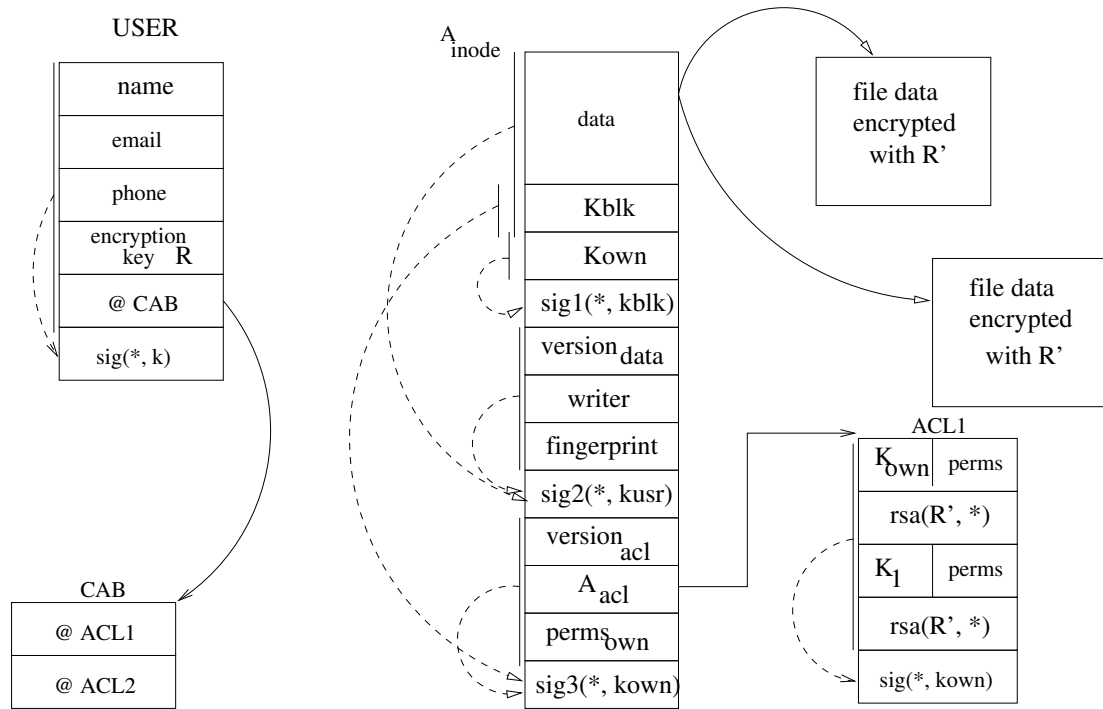


FIG. 2.6 – utilisation du CAB

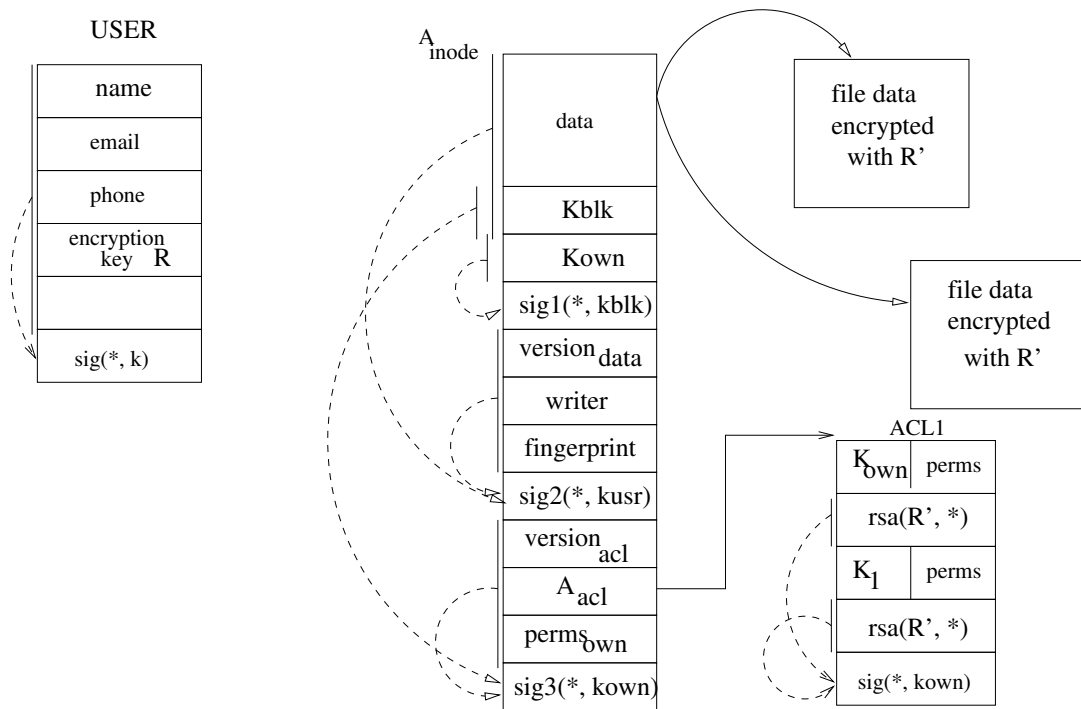


FIG. 2.7 – signature des clefs

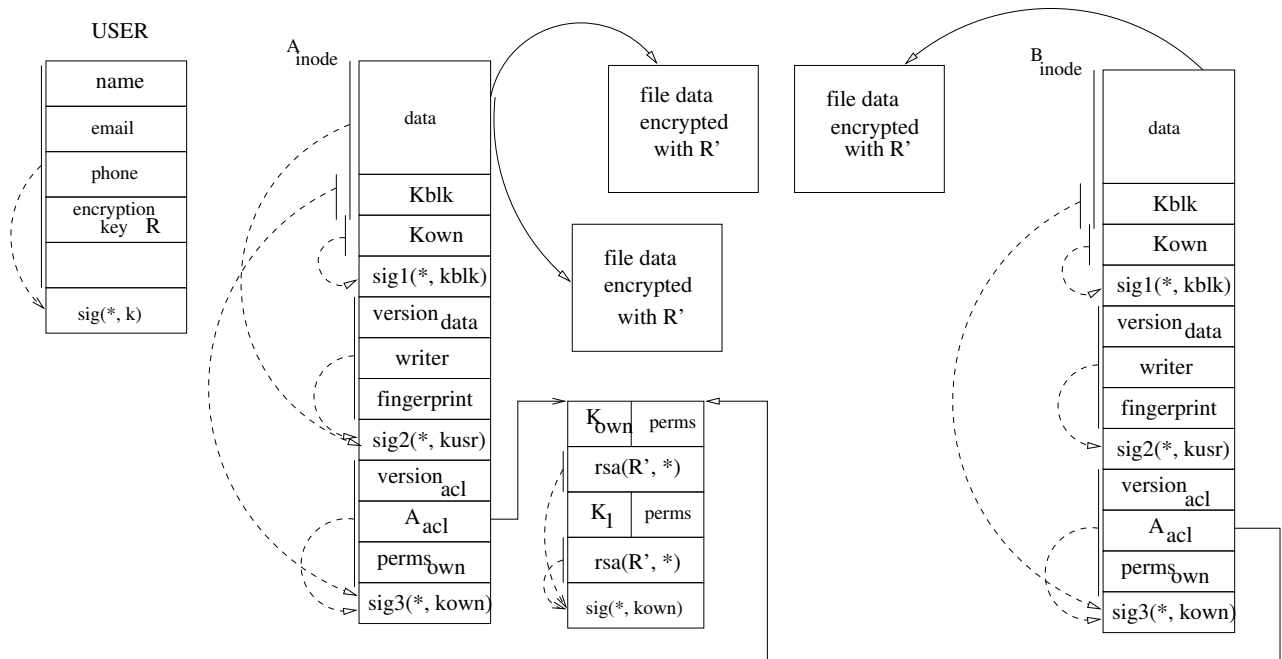


FIG. 2.8 – partage d'un ACL

Suppression du partage avec un lecteur

Dans le cas où nous voudrions supprimer le droit de lecture d'une personne, la démarche serait la suivante :

- * on simule l'ACL résultant de la suppression d'un lecteur
- * s'il existe déjà, nous allons réencrypter les données avec la clef contenue dans l'ACL existant puis mettre son adresse dans l'inode du fichier
- * s'il n'existe pas, on génère une nouvelle clef d'encryption et on crée un nouvel ACL pour le fichier puis on réencrypte les données avec la nouvelle clef

Le fait de réencrypter tout de suite le contenu du fichier peut provoquer une opération inutile dans le cas où le contenu du fichier ne change jamais (l'ancien lecteur aura toujours une copie valide).

Lecture d'un fichier

Pour lire le contenu d'un fichier, la méthode est la suivante :

- * on se localise dans l'ACL du fichier en utilisant le hash de sa clef publique
- * on décrypte la clef d'encryption du fichier en se servant de notre clef privée
- * on décrypte le contenu du fichier à l'aide de la clef obtenue

Ecriture dans un fichier

On rappelle que par hypothèse, un écrivain est aussi un lecteur, car il va avoir besoin de la clef afin d'encrypter ce qu'il a modifié.

Lors de la sauvegarde des modifications, on va récupérer la clef d'encryption de la même façon que pour lire un fichier, puis on va uniquement réencrypter les blocs que nous avons modifiés.

Avantages

L'avantage majeur de cette solution est la factorisation des ACL des fichiers que nous possédons ce qui va nous faire gagner la place nécessaire pour stocker les clefs d'encryption.

De plus, comme vu ci-dessus, une modification d'un bloc du fichier ne va pas demander une réencryption de tous les blocs même si la clef est commune à tout le fichier, car elle n'est pas basée sur le contenu de ce dernier.

Un autre avantage est que nous avons une grande probabilité, lors d'une session de travail, d'ouvrir plusieurs fichiers à la suite qui possèdent les mêmes droits. Nous allons donc pouvoir profiter du cache afin de stocker les ACLs et ainsi éviter de faire une requête sur le réseau pour le rapatrier.

Inconvénients

L'inconvénient de cette méthode est le fait que l'ajout ainsi que la suppression d'un lecteur soient gourmands en ressources. En effet, pour ces opérations, il est nécessaire de réencrypter tous les blocs du fichier et de les rapatrier par le réseau.

Nous pouvons néanmoins nous baser sur le fait que les utilisateurs changent très rarement les droits sur leurs fichiers ; les deux opérations problématiques ne seront donc pas effectuées très souvent.

2.1.3 Encryption asymétrique

En étudiant les deux solutions (convergente et ACL), on peut s'apercevoir qu'il y a un point qui diffère dans ce que l'on peut faire sur un système de fichiers sur un système de type UNIX.

Dans un tel type de système de fichiers, on peut très bien avoir un écrivain qui n'a pas le droit de lecture sur un fichier. Cela peut paraître étrange, mais on pourrait imaginer un script shell permettant d'écrire dans un fichier des logs à l'aide d'une double indirection, sans que la personne qui lance ce script puisse voir le contenu du fichier.

Une solution à ce problème serait de prendre un couple de clef privée et publique (ex. RSA) comme clef d'encryption à la place du hash. On donnerait alors la clef publique aux écrivains et la clef privée aux lecteurs. De cette manière, tout le monde peut utiliser le fichier en fonction de ses droits.

Avantage

L'avantage de cette solution est une bonne distinction entre les écrivains et les lecteurs.

Inconvénients

Il y a deux inconvénients majeurs dans cette solution :

Le premier est la perte de ce qui fait la force de l'encryption convergente, à savoir la factorisation des blocs de données encryptés.

Le second est la perte de temps causée par l'utilisation d'une encryption asymétrique (ex. RSA) à la place d'une encryption symétrique (ex. AES), la cryptographie asymétrique étant beaucoup plus coûteuse en ressources système.

2.1.4 Etude du système de fichiers du LIP6

Afin de stocker leurs données, les personnes travaillant au LIP6 ont accès à un serveur NFS qui contient leurs dossiers personnels (/home sous Unix). Il se peut quelque fois que des personnes donnent un accès en lecture ou en écriture à des collègues afin de travailler sur un projet commun.

L'étude d'un tel système nous a permis d'avoir un pool de fichiers nous permettant de regarder quels sont les droits les plus utilisés, combien de fichiers sont partagés avec les mêmes droits (ce qui correspond au nombre d'ACL dans *Pastis*). Mon maître de stage a donc en collaboration avec l'un des administrateurs réseau du groupe SRC, créé un fichier regroupant toutes les données sur le système de fichiers.

Mon travail fut alors d'écrire un script qui permettrait d'avoir des informations précises tirées des 200Mo de logs. Pour ce faire j'ai choisi le langage PERL qui est adapté au traitement de fichiers textes.

Le résultat de cette étude sur un système de fichiers, regroupant 1564102 fichiers et dossiers pour 122 utilisateurs, nous donne que les droits les plus courants est le mode 0644 (lecture et écriture pour le propriétaire et lecture pour les membres de son groupe et les autres). Concernant les nombres de tuples utilisateur, groupe, nous en avons 190 différents et 549 tuples utilisateur, groupe, droits pour le groupe différent. Le nombre moyen d'entrées par dossiers est de 8 fichiers/dossiers. Les fichiers ayant une taille moyenne de 75Ko.

Cette étude nous fut utile afin de confirmer notre choix de solution. Sur la page suivante, les résultats obtenus par le script :

-----resultats-----

Nombre d'entrees: 1564102
Mode le plus utilise: 0644
Nombre de modes differents: 60
Nombre d'utilisateurs: 122
Nombre de tuples (uid, gid): 190
Nombre de tuples (uid, gid, group_mode): 549

Fichiers:
* nombre: 1369827
* taille moyenne: 76879 octets

Dossiers:
* nombre: 182820
* nombre d'entrees en moyenne: 8

Liens symbolique:
* nombre: 7255

Liens:
* nombre: 4200

2.1.5 Solution choisie

La solution retenue afin de mettre en place un contrôle d'accès en lecture dans *PAS-TIS* va être la solution qui utilise les ACLs.

La solution qui utilise l'encryption convergente va être trop coûteuse en espace sur le système de fichiers. Suite à des tests, nous avons vu que nous allons plutôt nous retrouver avec un taux de similitude de 15% et non 30% comme annoncé par *Microsoft Research*. De ce fait, la place prise par les méta-données des EKB n'est plus compensée par celle gagnée par le fusionnement des blocs de données. De plus, cette solution nécessitait un EKB par bloc de données ce qui nous demande de récupérer beaucoup plus de données par le réseau et donc nécessite une plus grande bande passante.

D'autre part, nous avons décidé de ne pas empêcher un écrivain d'être aussi lecteur par défaut, car les cas où nous allons avoir le droit en écriture mais pas en lecture étant trop peu nombreux. De plus, l'utilisation d'une encryption asymétrique nous aurait été trop coûteuse en ressources.

Le seul défaut de la solution choisie est donc le coût de l'ajout et de la suppression d'un lecteur. Néanmoins, nous partons avec l'hypothèse que les changements de droits ne s'effectueront pas très souvent, ce qui nous permet de minimiser l'impact de cet inconvénient.

2.1.6 Tableau de comparaison entre EKB et ACL

fichiers	taille moyenne (Mo)	taille bloc(Ko)	% blocs similaires	(*)
10 000	3	64	15	80

(* : utilisateurs partageant le même fichier)

cas	encryption convergente	ACL
nombre de blocs	480 000	480 000
nombre de blocs encryptés	408 000	48 000
gain (Mo)	4 500	0
place prise par les EKB (Ko)	4 717 500	0
place prise par les ACL (Ko)	2.69	18.31
place prise par les fichiers + métadonnées (Mo)	30 106.93	30 000.02

Une fois ma solution validée par mon maître de stage, je suis passé à la partie implémentation de mon stage.

2.2 Implémentation

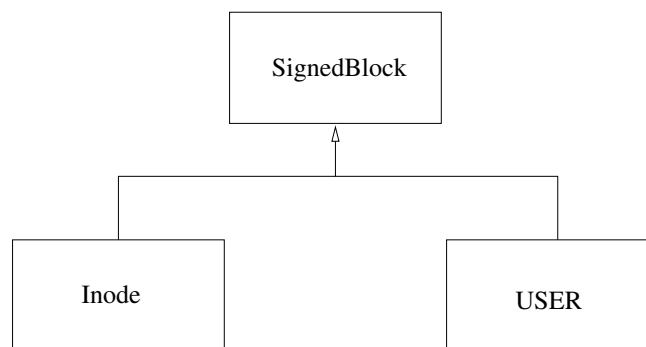
Nous allons à présent voir la façon dont la solution choisie a été implémentée.

2.2.1 Ajout du bloc USER

Comme vue précédemment, la solution retenue se base sur le bloc USER afin de récupérer la clef d'encryption pour les fichiers dont nous sommes le seul à avoir accès.

Ce bloc avait été proposé lors de l'implémentation des droits en écriture, mais n'avait pas été mis en place. J'ai donc repris l'étude en question pour pouvoir implémenter ce nouveau type de bloc.

Le bloc USER est un bloc qui peut être modifié au cours du temps (ex : changement d'adresse e-mail), c'est pourquoi, il fait partie des blocs signés (PKB, qui ont pour identifiant dans la DHT leur clef publique). Nous obtenons alors le diagramme UML suivant :



Du fait que User dérive de SignedBlock, le code contenu dans la classe Java est très simple car tout ce qui concerne l'encapsulation du bloc pour l'insertion dans la DHT va être réalisé par SignedBlock.

Voici un exemple avec le code du constructeur de la classe User :

```
public User(PublicKey pubkey)
{
    super ();
    setBlockKey (pubkey) ;
    setOwnerKey (pubkey) ;
}
```

L'appel à `super ()` va initialiser les attributs hérités de la classe mère, puis on utilise la clef publique de l'utilisateur à qui appartient ce bloc comme identifiant du bloc dans la DHT.

2.2.2 Modification du gestionnaire de sécurité

Afin de mettre en place le contrôle en lecture, il faut que nous encryption les données des fichiers.

Dans *PASTIS*, tout ce qui touche à l'encryption est géré par le gestionnaire de sécurité (correspondant à la classe `SecMan`). Il a donc fallu que j'ajoute des méthodes afin d'encrypter et décrypter une clef symétrique (DES) avec des clefs asymétriques (RSA), ainsi que des blocs de données avec une clef symétrique.

J'ai donc rajouté deux méthodes pour la gestion des clefs :

```
* public static byte[] encryptSecretKey(SecretKey seckey, PublicKey pubkey)
* public static SecretKey decryptSecretKey(byte[] seckey, PrivateKey privateKey)
Ainsi que deux méthodes pour l'encryption et la décryption des données :
```

```
* public static byte[] encryptData(byte[] data, SecretKey seckey)
* public static byte[] decryptData(byte[] encdata, SecretKey seckey)
```

Les classes `SecretKey` et `PublicKey` utilisées font partie de la bibliothèque `javax.crypto` qui est utilisée pour toute la partie cryptographique de *PASTIS*.

2.2.3 Mise en place dans le code existant

Une fois les classes utilisées modifiées, il faut remanier le code des fonctions qui vont être appelées lorsqu'une lecture ou une écriture va être effectuée sur un fichier afin d'y ajouter l'encryption et la décryption.

Utilisation de *callbacks*

Lors de la prise en main du projet, ce fut la partie la moins facile, le code de *PASTIS* étant écrit à l'envers.

En effet, lorsque l'utilisateur va faire un appel système à la fonction `read` par exemple, nous allons devoir aller chercher le bloc de données correspondant sur le réseau formé par les noeuds *Pastry*. Nous passons alors par *Past* qui lui n'utilise que des *callbacks* et non pas une valeur de retour pour ses méthodes. Cette méthode est utilisée afin de permettre une exécution asynchrone du code, ce qui est un avantage.

Ceci nous oblige donc à créer un objet de type *callback* qui doit avoir une méthode `void ReceiveResult(Object obj)` ainsi qu'une méthode `void ReceiveException(Exception e)` qui vont être appelées soit à la fin de la fonction de *Past* soit si une erreur se produit.

Nous n'écrivons ainsi pas le code dans l'ordre d'exécution, mais tout d'abord l'objet *callback* et ensuite le code qui va utiliser cet objet. Voici le squelette d'une classe utilisée comme continuation :

```
private class InitializeCont extends PastisCB
{
```

```

public void receiveResult(Object o)
{
    ...
}
public void receiveException(Exception e)
{
    ...
}
}

```

Cet objet de type continuation une fois instancié, sera passé en paramètre à une méthode de la couche *Past*.

Du fait que le code va être écrit en une succession de continuations, nous ne pouvons pas utiliser les mêmes variables d'une continuation à l'autre car le seul paramètre qui est passé à `receiveResult()` est le bloc que nous avons demandé à *Past*.

Pour passer outre ce problème, les concepteurs de *PASTIS* ont pourvu la classe `PastisCB`, utilisée pour les continuations d'un attribut, appelé contexte, qui va contenir toutes les variables dont nous avons besoin au cours des continuations.

Avant de passer une continuation à *Past*, nous allons donc lui associer un contexte. La classe représentant les contextes peut être redéfinie afin de satisfaire les besoins de l'utilisateur de la continuation.

Par exemple, le contexte utilisé pour l'appel à `write()` est :

```

private class WriteCtx extends Ctx
{
    Handle handle;
    DataBlock block;
    FileInode inode;
    List blist;
    Authinfo ainfo;
    int mode;
    SecretKey secret;
}

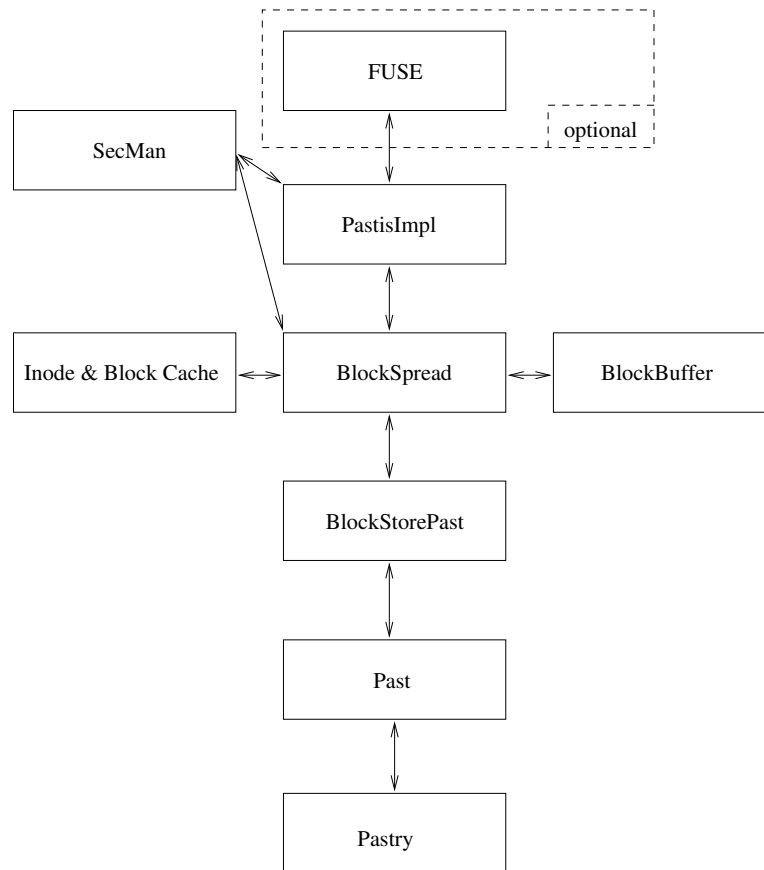
```

Nous allons nous servir des méthodes `getCtx()` et `setCtx()` des continuations afin de récupérer et changer le contexte de ces dernières.

Modifications pour la lecture et l'écriture

Les méthodes à modifier (`read()` et `write()`) se trouvent dans `PastisImpl.java`. Les blocs utilisés proviennent de `BlockSpread.java` qui se charge de distribuer les blocs entre *Pastis*, *Past* et les deux caches (un pour les inodes et un pour les blocs de données) qui servent à optimiser la récupération de blocs par *Pastis*.

Ci-dessous, une représentation de l'architecture de *PASTIS* :



Les fonctions cryptographiques sont très coûteuses en temps processeur, il faut donc éviter leur utilisation au maximum afin qu'il n'y ait pas d'impact sur les performances du système de fichiers.

Mon approche a donc été d'utiliser les caches afin d'y stocker les données en clair et ainsi, d'effectuer des opérations cryptographiques uniquement quand les caches sont vidés dans la DHT.

Pour ce faire, lors d'un appel à `read()` ou `write()`, on va récupérer l'ACL correspondant au fichier, puis la clef d'encryption de l'utilisateur afin de la passer en paramètre à la fonction qui va chercher le bloc de données demandé. Si le bloc est dans le cache, on récupère directement les données. Sinon, on passe la clef à la méthode qui va mettre le bloc dans la DHT afin que ce soit elle qui encrypte ou décrypte les données.

C'est lors de la récupération des ACLs que j'ai donc dû utiliser les continuations, car un ACL ne fait pas partie d'une inode, mais est un bloc à part que nous devons récupérer par le biais de *Past*.

2.2.4 Modifications pour intégrer les ACLs

Les ACLs avaient été implémentés mais pas encore utilisés dans *PASTIS* car l'interface *NFS* qui sert à utiliser *PASTIS* sur un système n'implémente pas les fonctions qui permettent de modifier des ACLs.

Les ACLs sont liés à un fichier et sont, sur les systèmes de fichiers (UNIX ou Windows), stockés dans les attributs étendus du fichier. Ces attributs sont utilisables via les deux appels système `getxattr()` et `setxattr()` (pour Get/Set eXtended Attributes).

Il a tout d'abord fallu que je fusionne ma version de *PASTIS* avec celle qui utilise *FUSE* (*Filesystem in UserSpace*) comme interface avec le système à la place de *NFS*.

Une fois ce travail effectué, j'ai dû modifier l'interface qu'avait *PASTIS* avec *FUSE* afin d'utiliser une autre version de ce dernier qui gère les attributs étendus. Modifier l'interface avec *FUSE* me fut facile car j'avais acquis une bonne connaissance de cet outil durant l'année. En effet, mon principal projet au LSE (Laboratoire Systèmes d'Epita) était de réaliser le portage de cette application qui fonctionne dans le noyau *Linux* vers le noyau de *NetBSD*.

Je me suis rendu compte lors d'une étude des commandes du shell qui permettent de modifier les ACLs d'un fichier que leur façon de modifier les ACLs n'est pas compatible avec la façon dont *PASTIS* fonctionne. Cette étude fut réalisée avec la commande *strace* qui permet de voir quels sont les appels système que va faire un processus. Ces commandes appellent la fonction `setxattr()` avec comme valeur de champs `system.posix_acl_access` et donnent le nom de l'utilisateur et les nouveaux droits comme valeur. Dans notre implémentation, l'appel devait se faire avec le nom de l'utilisateur comme nom de champs et les nouveaux droits comme valeur. J'ai donc réalisé un programme en C qui permet de manipuler les ACLs pour *PASTIS*. Ce programme prend en arguments sur la ligne de commande le chemin vers le fichier, le nom de l'utilisateur, les droits valides pour cet utilisateur ainsi que le mode de fonctionnement (récupérer les droits ou mettre des droits à un utilisateur).

Nous fûmes alors confrontés à un autre problème : dans *PASTIS* les utilisateurs sont reconnus par leur clef publique et c'est grâce à elle que nous pouvons par exemple les trouver dans un ACL.

Afin de pouvoir passer directement un nom pour l'accès aux ACLs, j'ai ajouté un nouveau bloc, dérivant de CHB car ne devant pas être modifiable, qui permet de faire la correspondance entre un nom et une clef publique.

Il y a néanmoins un problème avec cette méthode : si un attaquant crée de faux blocs de transition en y mettant sa clef, nous allons l'ajouter lui, et non la personne à qui le nom correspond.

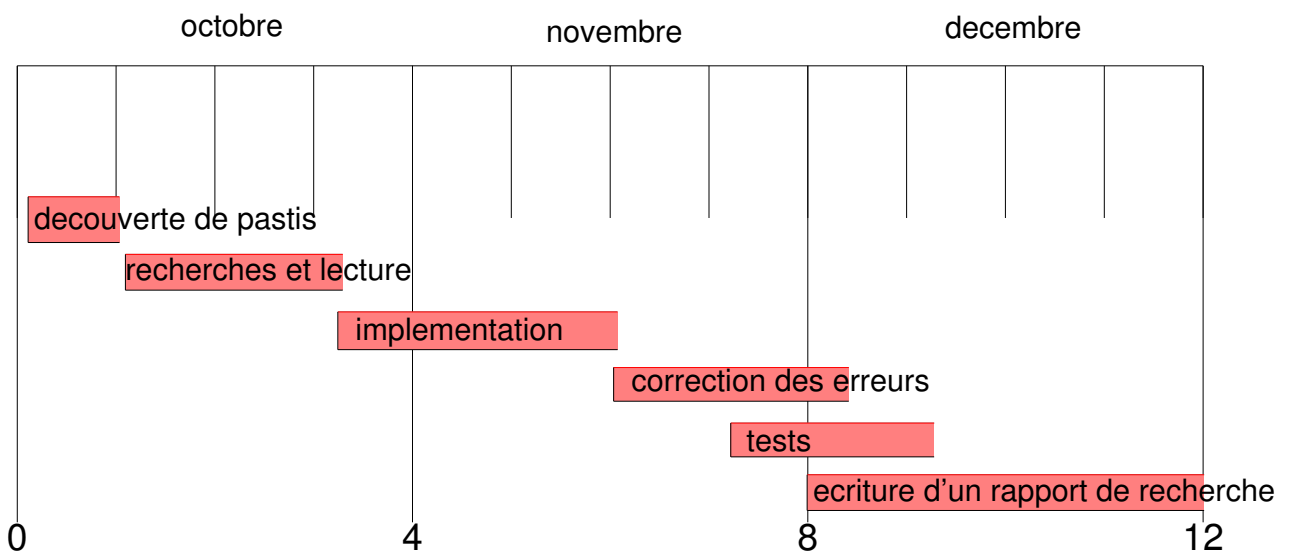
La partie la plus importante fut la correction d'erreurs, car il est connu en informatique qu'il faut au moins le double du temps mis pour écrire un programme pour en corriger les erreurs.

Comme je l'ai évoqué auparavant, *PASTIS* est bâti en couches (Pastis, puis Past et enfin Pastry). En cas de modification dans la couche la plus haute, on peut très bien provoquer une erreur dans les niveaux inférieurs car on a modifié les propriétés d'un objet sans le lui indiquer. Il devient alors très complexe de trouver la cause d'une erreur.

Les options que j'ai implémentées sont bien entendu désactivables afin de faire des tests de comparaison. Le tout se faisant par l'utilisation de l'option `-cipher 1` pour les activer ou `-cipher 0` pour les désactiver.

2.2.5 Déroulement du projet

Ci-dessous, un chronogramme reprenant le déroulement de mon stage :



Nous allons maintenant passer à la présentation des tests que j'ai utilisé lors de l'implémentation du contrôle d'accès en lecture.

2.3 Tests

Afin de valider le comportement de *Pastis* au fur et à mesure de mes modifications, j'ai développé ce script de test :

```
echo_color "Test for PASTIS encryption implementation" $lightblue
echo_color "_____ " $lightblue
echo

rm -rf /mnt/pastis/*

ref=`ls ../src/acltool/`

echo_color "Step1: copy files to /mnt/pastis" $lightblue
for i in $ref; do
    name="../src/acltool/"$i
    cp $name /mnt/pastis/
done

echo_color "Step2: check if the copies are correct" $lightblue
succ=0
cpt=0
ref=`ls ../src/acltool/`
for i in $ref; do
    cpt=`expr $cpt + 1`
    name="/mnt/pastis/"$i
    tmp="../src/acltool/"$i
    d=`diff $tmp $name`
    if ! [ -e "$name" ]; then
        echo "$i-->"
        echo_color "[FAIL]" $red
    else
        if [ -z "$d" ]; then
            succ=`expr $succ + 1`
            echo "$i -->"
            echo_color "[PASS]" $green
        else
            echo "$i-->"
            echo_color "[FAIL]" $red
        fi
    fi
done
echo_color "_____ " $lightblue
echo
if [ "$succ" -eq "$cpt" ]; then
    echo_color "Result: $succ out of $cpt are OK" $green
else
    echo_color "Result: $succ out of $cpt are OK" $red
fi
```

§

Ce script est à exécuter une fois *Pastis* lancé avec *fusermount* ou *nfsserver*.

Nous allons tout d'abord "nettoyer" le contenu du point de montage, *Pastis* ne se montant que dans un dossier vide.

Ensuite, nous allons copier les fichiers qui vont nous servir à tester le bon fonctionnement du système.

Une fois cette copie réalisée, nous allons tester s'il y a des différences entre le fichier qui se trouve sur le système de fichiers local et la copie qui se trouve sur *Pastis*.

J'ai également utilisé le "Andrew Benchmark" en tant que test de validation avec l'option `-verify`. Ce test consiste à copier les sources d'un programme vers le répertoire destination (dans notre cas `/mnt/pastis`), puis de les compiler et d'exécuter les binaires résultants.

```
./andrew -src ../tests/abdir -dest /mnt/pastis -verify
```

2.4 Mesures de performances

Une fois l'implémentation finalisée, j'ai effectué des tests de performances afin d'évaluer le coût de la gestion de l'accès en lecture sur le système de fichiers.

2.4.1 Ecriture et lecture de blocs

Le premier test consiste à comparer le temps que mettent l'écriture et la lecture de blocs de 1Ko (1024 octets) et de 1Mo (1024*1024 octets).

Afin de réaliser ce test, j'ai utilisé les commandes suivantes :

```
time python -c 'print "'A'" * (8 * 1024)' > /mnt/pastis/test
time python -c 'print "'A'" * (1024 * 1024)' > /mnt/pastis/test
time cat /mnt/pastis/test
```

Nous obtenons alors les résultats suivants en désactivant l'encryption :

Opération	Temps (seconde)
lecture 8Ko	0.175
écriture 8Ko	0.445
lecture 1Mo	1.958
écriture 1Mo	8.697

Puis les résultats suivant une fois l'encryption activée :

Opération	Temps (seconde)	Ajout
lecture 8Ko	0.391	+123%
écriture 8Ko	2.068	+364%
lecture 1Mo	2.750	+40%
écriture 1Mo	12.357	+42%

On peut voir que les temps de lecture ne diffèrent pas beaucoup alors que les temps pour l'écriture ont une différence assez grande.

Les temps important à l'écriture sont dû au fait que *FUSE* nous retransmet des appels à *write()* en une multitude d'appels pour des blocs de 4096 octets et ceci même si l'appel système fait depuis l'application était beaucoup plus important. Nous allons donc devoir faire beaucoup plus de fois le chemin entre l'appel de *FUSE* et la restitution du bloc que lors des appels à *read()* qui eux étaient de 1Ko à chaque fois.

2.4.2 Comparaison avec d'autres systèmes de fichiers

Il n'est pas aisé de comparer *PASTIS* à d'autres systèmes de fichiers répartis large échelle, pour la raison qu'il est l'un des rares à implémenter un contrôle d'accès en lecture. Les autres systèmes sont soit non implémentés (sous forme de rapports uniquement), soit délèguent le contrôle en lecture à la couche supérieure, c'est à dire aux applications l'utilisant.

J'ai donc retenu deux systèmes de fichiers pour réaliser cette comparaison. L'un d'eux reposant lui aussi sur *FUSE*.

SSHFS

Le premier est donc *sshfs* qui repose sur *FUSE*. Le principe de ce système de fichiers est le suivant : il permet d'accéder à une partie des fichiers d'une machine distante via une connection *ssh*.

Comme *Pastis*, il utilise donc *FUSE*, mais aussi l'encryption et une connection réseau. Les temps des différentes opérations sont de l'ordre de la milliseconde si on enlève la latence réseau. Il n'est donc ralenti que par la qualité de la liaison entre les deux machines.

Néanmoins, ce système ne permet que de connecter deux machines entre elles et n'intègre pas la réplication des blocs afin d'assurer l'intégrité des données.

CNFS

Ce système de fichiers est le résultat d'une étude menée par le département d'informatique du *Trinity College Dublin*. Cette étude vise elle aussi à implémenter un contrôle d'accès dans un système de fichiers distribué.

Ils donnent les résultats suivants pour leurs tests :

Opération	Temps (milliseconde)
lecture 8Ko	0.63
écriture 8Ko	23.062
lecture 1Mo	874.768
écriture 1Mo	1480.16

Il semblerait que leur système de fichiers soit une modification de NFS, d'où leurs résultats. Ils utilisent les performances de NFS, alourdi par leur système d'encryption (encryption symétrique des données et asymétrique des méta-données). Leur système est donc centralisé et ne gère pas la réplication des blocs.

Nous pouvons donc dire en conclusion de cette partie sur les comparaisons avec les autres systèmes utilisant l'encryption, que certes *Pastis* est plus lent, mais il a été implémenté en Java et non en C ce qui lui fait perdre un peu de rapidité. Mais aussi, *PASTIS* est un système qui gère entièrement la sécurité des données, par l'encryption et aussi la réplication qui protège des pertes ou modifications.

Si on le compare avec des systèmes qui font de la réplication mais pas d'encryption (comme *Ivy* et *OceanStore* par exemple), les précédentes études laissent penser que *PASTIS* est plus rapide.

Il n'est donc pas facile de comparer des systèmes qui sont si différents et où beaucoup de paramètres peuvent influencer leur comportement (comme la taille du cache ou des blocs de données par exemple).

Chapitre 3

Conclusion

Mon travail en commençant ce stage était de rechercher et d'implémenter une solution afin de gérer le contrôle d'accès en lecture sur un système de fichiers distribué.

Avant que je ne commence, le système était déjà utilisable mais pas de façon entièrement sécurisée. Un précédent stagiaire avait renforcé la gestion des droits d'accès en écriture sur ce système de fichiers, mais la lecture n'avait pas encore été abordée.

J'ai donc commencé par étudier une solution développée par *Microsoft Research*, l'encryption convergente. Elle consiste à fusionner les blocs de données similaires afin de gagner de la place, puis de les encrypter. Au terme de l'étude de cette solution, je me suis rendu compte qu'elle ne correspondait pas entièrement au besoin de *PASTIS* car elle nécessite beaucoup trop de place supplémentaire si le taux de similitude des blocs de données n'atteint pas une certaine valeur.

Je me suis donc basé sur l'étude qu'avait menée la personne ayant implémenté les droits d'accès en écriture afin de créer une solution pour l'accès en lecture.

Suite à un test réalisé par Jean-Michel Busca, les résultats obtenus et traités par un script PERL nous ont permis de nous conforter dans le choix de la solution. L'implémentation ne fut pas facile. L'écriture du code supplémentaire était aisée, mais la recherche d'erreurs fut plus complexe. Ceci était dû à l'utilisation de certaines méthodes comme les *callbacks*, mais aussi à certaines parties du code qui auraient pu être simplifiées afin de faciliter la recherche d'erreurs.

Une fois l'implémentation réalisée, il fallut tester le bon fonctionnement de *Pastis* afin de s'assurer que mes modifications n'avaient pas modifié le comportement du système.

La dernière partie fut la réalisation de tests de performances et de comparaison

avec d'autres systèmes de fichiers. Cette partie fut complexe et pas nécessairement concluante, du fait de la spécificité de *Pastis* et de la multitude de paramètres qui peuvent influencer sur les performances d'un système de fichiers.