Pastis : un système de fichiers pair à pair multi-écrivain passant l'échelle

Jean-Michel Busca[2] Fabio Picconi[1] Pierre Sens[2]

[1]LIP6, Université Paris 6 - CNRS, Paris, France [2]INRIA, Rocquencourt, France

Résumé

Nous présentons Pastis, un système de fichiers pair à pair en lecture - écriture d'architecture entièrement décentralisée et supportant de très nombreux utilisateurs. Pastis utilise pour représenter un système de fichiers une structure similaire à celle d'un système de fichiers Unix, composée de blocs inode et de blocs de données. L'ensemble de ces blocs est stocké dans la table de hashage distribuée Past, construite au dessus du réseau pair à pair structuré Pastry.

Les propriétés de tolérance aux fautes et de localité de Past et Pastry fournissent à Pastis un support de stockage hautement disponible et performant. Par la mise en œuvre de modèles de cohérence appropriés, Pastis tire parti de ces propriétés encore améliorer les performances en maintenant les coûts d'accès réseau à un niveau minimum. Pastis garantit par ailleurs l'authenticité et l'intégrité des données stockées, au travers de mécanismes standards de cryptographie.

Nous avons développé un prototype de façon à évaluer les performances de notre architecture. Ce prototype est codé en Java et utilise FreePastry, l'implémentation open-source de Past et Pastry. Au travers d'une interface d'accès propriétaire, il ffre dans sa version actuelle aux applications le choix entre deux modèles de cohérence. Les premières mesures de performances suggèrent que Pastis n'est qu'à peine deux fois plus lent que NFS à configuration comparable.

1 Introduction

De nombreux systèmes de fichiers pair à pair ont récemment été proposés [3, 8, 9, 11, 14, 15], mais peu semblent pouvoir être déployés sur des centaines de milliers de nœuds tout en offrant un accès en lecture et écriture à un très grand nombre d'utilisateurs. De plus, peu de prototypes ont été développés à ce jour, et les mesures de performances manquent pour valider et comparer les différentes architectures proposées.

De fait, maintenir la cohérence et assurer la sécurité des données lors de mises à jour, tout en conservant de bonnes performances lorsque le système atteind une très grande échelle, est une tâche difficile. Les systèmes de fichiers en lecture seule, tels que CFS [14], sont plus simples à concevoir dans la mesure où ils supposent que les données du système sont rarement, voire jamais mises à jour. Ceci permet de mettre en œuvre une politique de caching intensif, les données cachées étant rarement invalidée et conservées jusqu'à expiration. L'intégrité des données peut être vérifiée facilement au travers de fonctions de hashage et de la signature du répertoire racine par l'administrateur du système. Enfin, la question de la cohérence des données ne se pose pas dans la mesure où seul l'administrateur du système de fichiers est autorisé à le modifier.

Les architectures multi-écrivain doivent résoudre plusieurs problèmes qui ne se posent pas dans les systèmes en lecture seule. Il s'agit notamment de maintenir la cohérence des différentes répliques d'une donnée, assurer le contrôle d'accès aux fichiers, authentifier les requêtes de mise à jour et gérer les conflits d'accès.

Le système Ivy [11], par exemple, matérialise le contenu du système de fichiers par un ensemble de journaux, un par utilisateur. Ces journaux sont stockés dans la table de hashage distribuée (Distributed Hash Table, ou DHT) DHash, elle-même implémentée sur le réseau pair à pair structurée (Key Based Routing, ou KBR) Chord [13]. Lorsqu'il modifie le système de fichiers, un utilisateur ajoute en tête de son journal un enregistrement indiquant la modification effectuée et son estampille. Lors de la lecture d'un fichier, les journaux des différents utilisateurs sont parcourus à la recherche des dernières modifications apportées au fichier. Ce mécanisme présente le grand avantage d'éviter toute coordination des utilisateurs lors d'écritures concurrentes sur un même fichier. Par ailleurs, chaque utilisateur ne pouvant écrire que dans son propre journal,

toutes les modifications sont de facto authentifiées. En revanche, les opérations de lectures nécessitent de parcourir autant de journaux qu'il y a d'utilisateurs autorisés à écrire sur le système de fichiers. Ainsi, bien qu'Ivy puisse être déployé sur un très grand nombre de nœuds, il ne passe pas l'échelle sur le nombre d'écrivains.

OceanStore [9] s'appuie pour stocker ses données sur Tapestry, un système de localisation et de routage décentralisé (Decentralized Object Location and Routing, ou DOLR) permettant une plus grande souplesse dans le placement des données. Il adopte une approche différente d'Ivy quant au traitement des mises à jour en introduisant un certain degré de centralisation dans le système. Un sous-ensemble des nœuds, le *primary tier*, est chargé de sérialiser les requêtes de mise à jour provenant des autre nœuds, en utilisant l'algorithme de réplication de machine à état tolérant les fautes byzantines BFT [12]. La mise en œuvre de BFT étant coûteuse, les nœuds du *primary tier* doivent être hautement disponibles et fortement connectés, et dans l'esprit des concepteurs d'OceanStore, ces nœuds sont des serveurs dédiés, configurés et maintenus par des fournisseurs de services payant. Aussi, OceanStore n'est pas vraiment adapté à une communauté d'utilisateurs souhaitant utiliser un système indépendant de toute autorité centrale.

Pangaea [15] n'utilise ni DHT ni DOLR pour stocker ses données, et applique au contraire une politique de réplication agressive, où chaque utilisateur accédant à un fichier en crée une copie sur son nœud local. Cette politique a pour but de maximiser les performances d'accès en lecture, mais soulève deux difficultés majeures : la localisation des différentes répliques d'un fichier, et la propagation efficace des mises à jour. Pour les résoudre, Pangaea connecte les différentes répliques d'un fichier en un graphe aléatoire, dont la gestion est entièrement décentralisée. Ce graphe est utilisé pour propager les mises à jour en au moyen d'un algorithme optimisé d'inondation à deux phases, utilisant des messagers (harbingers). Bien qu'optimisé, ce mécanisme est cependant suceptible de générer un trafic important, lié au nombre des répliques d'un fichier. Par ailleurs Pangaea n'offre que peu de garanties quant à la cohérence et à la visibilité de ces mises à jour.

Dans ce contexte, nous avons conçu le système de fichiers pair à pair Pastis, qui lève les limitations des systèmes existants tout en conservant une architecture simple. Entièrement décentralisé, Pastis s'appuie sur la DHT Past et le KBR Pastry pour localiser efficacement les données et méta-données du système de fichiers. Il utilise pour représenter le système de fichiers une structure similaire à celle du système de fichiers Unix (UFS), et peut supporter de ce fait un très grand nombre d'utilisateurs en lecture - écriture. Il propose dans sa version actuelle deux modèles de cohérence clairement définis, qui permettent aux applications de maîtriser la sémantique de leurs accès aux fichiers. Ces modèles de cohérence s'appuient sur les propriétés de localité du support de stockage sous-jacent pour optimiser les accès réseau et augmenter les performances du système. Enfin, Pastis prend en charge la protection de l'intégrité des données stockées en proposant un mécanisme générique sur lequel peuvent être bâtis différents modèles de partage des fichiers.

Dans la suite de cet article, le paragraphe 2 présente l'infrastructure logicielle sur laquelle s'appuie Pastis. Le paragraphe 3 décrit l'architecture de notre système, en détaillant les points clés que sont les modèles de cohérence implémentés et la gestion de la sécurité des données. Le paragraphe 4 présente le prototype et l'environnement de test mis en œuvre. Le paragraphe 5 discute le résultat des différentes mesures de performances effectuées et le paragraphe 6 conclut.

2 Infrastructure

Pastis s'appuie pour stocker ses données sur le support stable fourni par la DHT Past, elle-même construite sur le KBR Pastry.

2.1 Routage et localité

Pastry [7] assigne aléatoirement à chaque nœud du réseau un identifiant unique sur n bits, compris entre 0 et $2^n - 1$, n valant typiquement 128. L'espace d'adressage ainsi constitué est considéré comme circulaire, l'identifiant 0 étant le successeur de l'identifiant $2^n - 1$. Pour router un message à travers le réseau, les applications utilisant Pastry lui associent une clé, clé dont l'espace des valeurs est le même que celui des identifiants de nœud. Le rôle de Pastry est alors de router ce message jusqu'au nœud dont l'identifiant est numériquement le plus proche de la clé du message. Pour ce faire, Pastry utilise un algorithme de routage dérivé de celui de Plaxton [1], et qui consiste à router le message de proche en proche, vers des nœuds partageant avec la clé de routage un préfixe de plus en plus grand. Ainsi, pour un réseau comprenant N nœuds, le routage d'un message ne nécessite au plus $log_{2^b}N$ sauts, 2^b étant la base de numération retenue pour les identifiants.

Outre son algorithme de routage efficace, la caractéristique distinctive de Pastry est qu'il permet la prise en compte d'un critère de distance entre les nœuds dans l'espace physique. Ce critère peut être quelconque, comme par exemple la latence des communications ou le débit disponible entre deux nœuds. Lors de la construction de la table de routage d'un nœud, Pastry renseigne systématiquement les entrées de la table avec les nœuds les plus proches du nœud local. Grâce à cette propriété de localité, le routage d'un message au travers de Pastry n'est qu'entre 30% à 40% plus long que le plus court chemin existant pour la métrique considérée.

2.2 Stockage et sécurité

Past [2] permet aux applications utilisatrices d'associer à un bloc de données quelconque une clé de stockage, que Past utilise pour stocker et retrouver le bloc de façon efficace au travers de Pastry. Past stocke un bloc de données sur le nœud, dit nœud *racine*, dont l'identifiant est numériquement le plus proche de la clé de stockage du bloc. De plus, de façon à fournir un stockage hautement disponible malgré la grande volatilité des nœuds du réseau, Past réplique le bloc sur les k nœuds adjacents au nœud racine dans l'espace d'adressage Pastry. De par la distribution aléatoire des identifiants de nœuds, ces k nœuds sont statistiquement localisés dans des zones géographiques très diverses, et présentent donc un taux de pannes corrélées très faible.

Il est important de noter que Past bénéficie d'une seconde propriété de localité de Pastry : lors de la recherche d'un bloc de données répliqué, la première réplique atteinte par Pastry est avec une grande probabilité celle qui est la plus proche du nœud ayant initié la recherche. Cette caractéristique est d'autant plus importante que, comme mentionné ci-dessus, la dispersion des répliques d'un même bloc dans l'espace physique est très grande. Pastis met à profit cette propriété de localité dans le modèle de cohérence read-your-writes décrit au paragraphe 3.3

Les nœuds constituant un réseau pair à pair et les données qu'ils stockent peuvent très facilement être l'objet d'attaques malveillantes. Il est donc essentiel d'implémenter des mécanismes permettant aux applications de vérifier l'intégrité des blocs qu'elles confient à Past. Pour ce faire, Past définit deux types de blocs : les Content Hash Bloc (CHB) et les Public Key Bloc (PKB). Les CHB sont utilisés pour stocker des données non modifiables. Par convention, leur clé de stockage est déduite du contenu même du bloc par l'application d'une fonction de hashage de type SHA-1 ou MD5. Les PKB sont utilisés pour stocker des données modifiables, et sont associés à un couple clé publique - clé privée RSA. Par convention, la clé de stockage d'un PKB est la clé publique associée au bloc, et le contenu du bloc doit être signé avec la clé privée correspondante.

Ainsi, lorsqu'une application demande à lire un bloc qu'elle identifie par sa clé de stockage, il lui est facile de vérifier que le bloc retourné par Past n'a pas été altéré. Dans le cas d'un CHB, l'application recalcule le hash du bloc et vérifie qu'il correspond bien à la clé de stockage ; dans le cas d'un PKB, l'application vérifie au moyen de la clé de stockage la validité la signature du bloc. Pour empêcher un utilisateur malveillant de remplacer des données déjà stockées par la simple utilisation de l'interface d'accès de Past, les contrôles réalisés par une application après la lecture d'un bloc sont également effectués par chaque nœud Past avant l'écriture d'un bloc.

L'efficacité de ces contrôles reposent sur les propriétés des fonctions de hashage et des clés asymétriques, qui garantissent qu'il est en pratique impossible à un utilisateur de générer un contenu correspondant à la clé de stockage d'un CHB donné, ou de correctement signer le contenu d'un PKB sans posséder la clé privée correspondante. Concernant la lecture de PKB, on note cependant que la validité de la signature du bloc lu n'implique pas que le contenu du PKB soit le dernier stocké : l'application doit mettre en œuvre un mécanisme supplémentaire pour s'en assurer, comme décrit au paragraphe 3.1.

3 Architecture

Pastis utilise pour représenter un système de fichiers une structure similaire à celle du système de fichiers Unix (UFS), comme le montre la figure 1. Chaque fichier est identifié par une structure de donnée de type inode, laquelle contient les métadonnées du fichier et est stockée dans un Public-Key Bloc (PKB) Past. La paire clé privée - clé publique associée à ce bloc est générée lors de la création du fichier par son propriétaire.

Un inode contient les différents attributs du fichier, ainsi que des informations relatives à la sécurité de son contenu. Les attributs du fichier sont ceux classiquement retournés par l'appel système Unix `stat()`, à l'exception des champs *uid*, *gid* et *mode*. Ces champs sont remplacés dans Pastis par les informations de

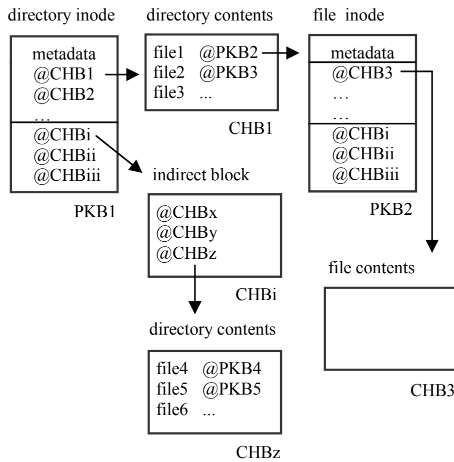

Fig. 1 – *Structure du système de fichiers*

sécurité qui permettent de gérer les droits d'accès d'un fichier de façon plus générale que le schéma Unix standard.

Le contenu d'un fichier est stocké dans des blocs de taille fixe, comme dans un périphérique de type bloc Unix. Dans Pastis cependant, les blocs de données sont non modifiables et stockés dans des Content Hash Bloc (CHB) Past. L'adresse de stockage d'un bloc correspond au hash du contenu du bloc, et cette adresse est stockée dans l'inode du fichier. Comme dans UFS, des blocs d'indirection double et triple sont utilisés pour stocker des fichiers de grande taille tout en conservant un inode de taille fixe.

Pastis représente un répertoire sous la forme d'un fichier de type particulier, dont le contenu liste les entrées du répertoire. Chaque entrée est décrite par un couple associant le nom du fichier à la clé de stockage Past de son inode. De façon à optimiser la lecture des répertoires, notamment lors de la résolution de longs chemins d'accès, un certain nombre d'entrées est directement stocké dans l'inode même du répertoire.

3.1 Lectures et mises à jour

Comme mentionné au paragraphe 2.2, Pastis doit à chaque lecture vérifier l'intégrité du bloc de données retourné par Past. Dans le cas d'un CHB, la vérification est aisée car la clé de stockage du bloc est liée à son contenu : une réplique du bloc retournée par Past dont le hash du contenu correspond à la clé de stockage est correcte. Dans le cas d'un PKB en revanche, les contenus successifs du bloc sont enregistrés sous la même clé de stockage, et vérifier l'intégrité d'une réplique de PKB ne suffit pas à garantir sa fraîcheur. Il est possible en effet qu'un ou plusieurs serveurs, sous le contrôle d'un utilisateur malveillant, retournent une version correctement signée du PKB mais ne correspondant pas à la dernière version enregistrée par Pastis.

Pour contrer ce type d'attaque, dite de *roll-back*, Pastis implémente le mécanisme suivant. Les versions successives d'un PKB sont identifiées par des estampilles strictement croissantes, enregistrées dans le PKB même. Lors de la lecture d'un PKB, Pastis demande à Past de lui retourner les k répliques disponibles de ce PKB. Il retient alors comme étant la plus récente la réplique dont le contenu est correctement signé et porte l'estampille la plus élevée. Ce mécanisme simple résiste jusqu'à $k-1$ attaques malveillantes, et utilise en fait la même estampille que celle servant à résoudre les conflits de mise à jour, décrits au paragraphe 3.2.

Ainsi, l'ouverture d'un fichier nécessite en général de lire l'ensemble des répliques de son inode (PKB) pour déterminer la plus récente. La lecture de chacun des blocs de données (CHB) constituant son contenu est en revanche beaucoup plus rapide : la lecture d'un CHB est terminée dès que la première réplique valide est retournée, et, statistiquement, il s'agit de la réplique la plus proche de l'utilisateur.

Lorsqu'il s'agit de mettre à jour un fichier, Pastis enchaîne les opérations suivantes :

1. *Lecture de l'inode du fichier depuis Past.*
2. *Lecture des blocs de données correspondant à la zone à mettre à jour.*
3. *Ecriture dans le cache local des nouvelles données, en créant de nouveaux CHB.*
4. *Insertion dans Past des CHB nouvellement créés. Le hash du contenu de chaque bloc doit être calculé pour déterminer l'adresse de stockage du CHB dans Past.*
5. *Mise à jour locale de l'inode pour refléter les nouvelles adresses de stockage des CHB. Si ces CHB sont accédés au travers de blocs d'indirection, ces derniers doivent également être mis à jour et réinsérés dans Past.*
6. *Mise à jour différée dans Past du PKB contenant l'inode, lors de la fermeture du fichier.*

Ainsi, à chaque mise à jour d'un fichier, de nouveaux CHB, contenant les données nouvellement écrites, sont insérés dans Past. Les anciens CHB, bien que n'étant plus référencés par le nouvel inode, ne sont pas pour autant supprimés de Past. Il est en effet nécessaire, lors de la mise en œuvre de modèles de cohérence relachés, de permettre à différents utilisateurs d'accéder à différentes versions du fichier. Cette méthode conduit néanmoins à un accroissement sans fin de l'espace de stockage utilisé par un fichier. Des travaux sont en cours pour améliorer ce point.

3.2 Conflits

Il se peut que plusieurs utilisateurs mettent simultanément à jour un inode en fermant le fichier correspondant après l'avoir modifié. Il faut dans ce cas assurer qu'en fin de mise à jour, toutes les répliques de l'inode sont identiques. Pour ce faire, nous mettons en œuvre un mécanisme de type *last writer wins*, s'appuyant sur des estampilles associées aux inodes.

Ce mécanisme fonctionne comme suit. Chaque inode contient une estampille identifiant son numéro de version et l'identité unique de l'utilisateur l'ayant modifié. La comparaison des numéros de version et, en cas d'égalité, des identifiants d'utilisateur définit un ordre total sur les estampilles. Avant d'enregistrer une nouvelle version d'un inode dans Past, l'utilisateur incrémente le numéro de version de l'inode lu, et fournit son propre identifiant. Chacun des nœuds Past stockant une des réplique de l'inode compare ensuite l'estampille du nouvel inode et celle de l'inode courant : le nouvel inode n'est enregistré que si son estampille est strictement supérieure à celle de l'inode courant.

Ce mécanisme simple est bien entendu compatible avec les modèles de cohérence supportés par Pastis, décrits au paragraphe suivant : en cas de mises à jour concurrentes, seule une des mises à jour est conservée. Il permet de plus de détecter les conflits et peut servir de base à la mise en place d'un mécanisme simple de sérialisation pour les fichiers dont le contenu est sensible, comme les répertoires.

3.3 Cohérence

Un modèle de cohérence définit des garanties de fraicheur et de cohérence des données lues, ainsi que des garanties de visibilité des données écrites. Il résulte d'un compromis entre les besoins réels d'une application et les performances du système de fichiers. Dans sa version actuelle, Pastis offre deux modèles de cohérence : le modèle close-to-open et le modèle read-your-writes.

Le modèle **close-to-open** est un modèle de cohérence relaché largement utilisé dans les systèmes de fichiers distribués [6, 10], et implémenté pour la première fois dans le système AFS. Ce modèle stipule que : (a) lorsqu'un utilisateur ouvre un fichier, il accède à la version la plus récente du fichier enregistrée par une fermeture, et (b) tant que l'utilisateur maintient le fichier ouvert, les seules modifications qu'il perçoit sont celles qu'il effectue.

L'avantage de ce modèle est qu'il garantit qu'à l'ouverture d'un fichier, l'utilisateur accède à une version cohérente de son contenu, que cette version est la plus récente, et qu'elle est stable tant que le fichier reste ouvert. En contrepartie, cela signifie que les mises à jour effectuées concurremment par différents utilisateurs sont isolées les unes des autres, et qu'à la fermeture du fichier, seule une de ces mises à jour sera retenue. L'utilisation de ce modèle fait l'hypothèse, vérifiée dans de nombreux environnements, qu'un fichier partagé en écriture est le plus souvent modifié par au plus un utilisateur à la fois.

Dans Pastis, le modèle close-to-open est implémenté comme suit : à l'ouverture d'un fichier, la version la plus récente de son inode est lue depuis Past et conservée localement jusqu'à la fermeture du fichier. Toutes les requêtes de lecture du fichier sont traitées en se référant aux adresses de CHB contenues dans la copie locale de l'inode. Quant aux requêtes d'écriture, elles se traduisent par la modification ou la création locales de blocs de données. Si ces blocs doivent être écrits sur Past faute de place sur la machine locale, ils ne sont de toute façon pas visibles par d'autres utilisateurs tant que la nouvelle version de l'inode n'est pas elle-même écrite dans Past.

Le modèle close-to-open garantit qu'à l'ouverture d'un fichier, la version la plus récente du fichier est accédée. Certaines applications n'ont cependant pas besoin d'une garantie aussi forte, et peuvent se contenter d'accéder à une version qui, sans être la plus récente, est au moins aussi récente que la précédente version générée localement. Ceci nous a conduit à développer un modèle de cohérence plus relaché que close-to-open, appelé read-your-writes.

Le modèle **read-your-writes** présente l'intérêt de réduire significativement le coût de lecture de l'inode d'un fichier. Il n'est en effet plus nécessaire de lire l'ensemble des répliques de l'inode pour s'assurer qu'on est bien en possession de la plus récente : il suffit de trouver une réplique dont l'estampille est au moins aussi récente qu'une estampille de référence. Deux facteurs concourent à réaliser cette recherche très rapidement : (a) la propriété de localité Pastry garantit avec une haute probabilité que la première réplique accédée est celle la plus proche de l'utilisateur, et qu'elle est donc retournée dans un temps minimal, (b) en l'absence de pannes ou de changements de configuration récents des nœuds stockant l'inode, toutes les répliques de l'inode sont à jour et satisfont le critère de recherche, dont notamment la première accédée.

Pastis implémente le modèle read-your-writes de la façon suivante. Chaque nœud maintient un cache contenant l'estampille de l'inode des fichiers écrits par l'uti-

lisateur local. Lorsque l'utilisateur ré-ouvre un fichier, Pastis recherche une réplique de son inode dont l'estampille est au moins aussi récente que celle contenue dans le cache. Cette recherche se déroule en deux phases : dans un premier temps, Pastis demande à Past de retourner la première réplique qu'il trouve dans le réseau. Si cette réplique ne satisfait pas le critère de recherche, Pastis demande dans un deuxième temps à Past de retourner l'ensemble des répliques et choisit la plus récente, comme pour le modèle close-to-open.

Si une entrée du cache des estampilles a été purgée faute de place entre le premier et le second accès au fichier, Pastis passe directement à la seconde phase de la recherche. Ceci ne remet pas en cause la sémantique read-your-writes de l'accès puisque la version du fichier stockée dans le réseau est nécessairement plus récente que celle écrite localement.

Le modèle read-your-writes aurait pu être implémenté différemment, par exemple en conservant localement l'inode d'un fichier au delà de sa fermeture, et en réutilisant ce même inode lors de l'ouverture suivante du fichier. L'implémentation retenue a l'avantage d'une part de minimiser la taille de cache nécessaire, et d'autre part de donner l'opportunité de lire, avec un coût additionnel réduit, une version plus récente du fichier.

La sémantique read-your-writes peut être facilement étendue à celle de *monotonic-reads* en consignant également dans le cache local l'estampille des fichiers qui ont été lus. Pastis peut ainsi, avec un coût très faible, garantir que la version lue lors de la réouverture d'un fichier est au moins aussi récente que celle lue précédemment.

3.4 Sécurité des données

Pastis est conçu pour s'exécuter sur une infrastructure non sûre, composée de nœuds appartenant à des utilisateurs anonymes et potentiellement malveillants. Les données stockées dans Pastis doivent donc être protégées contre toute modification frauduleuse (intégrité), et lorsque c'est nécessaire, contre toute lecture non autorisée (confidentialité).

3.5 Intégrité

Comme expliqué précédemment, l'inode d'un fichier est stocké dans un PKB Past, dont la paire clé publique - clé privée est générée par le propriétaire du fichier lors de sa création. Dans le mécanisme standard décrit paragraphe 2.2, toute modification du fichier nécessite de signer l'inode avec la clé privée du PKB, de façon à ce que l'intégrité de l'inode puisse ensuite être vérifiée lors de sa relecture. Le propriétaire d'un fichier qui souhaite le partager en écriture avec d'autres utilisateurs doit donc fournir à chacun de ces utilisateurs la clé privé du PKB pour qu'ils puissent signer leurs modifications.

Ce schéma pose cependant un problème : la dissémination de la clé privée accroit consédérablement les risques de divulgation accidentelle, quelque soit le soin apporté aux mécanismes de distribution et de stockage de cette clé. Or, en cas de divulgation de la clé, le propriétaire du fichier n'a d'autre ressource pour protéger le fichier que de le recopier sous un nouvel inode, associé à un nouveau couple clé privée - clé publique. Cette méthode va à l'encontre de la sémantique Unix standard qui identifie un fichier à son inode, et pose un problème pour tous les liens *hard* qui pointeraient vers le fichier.

Une solution à ce problème consiste à introduire une indirection entre un fichier et les utilisateurs autorisés à le modifier, sous la forme de certificats signés par le propriétaire du fichier. Dans cette solution, chaque utilisateur de Pastis est identifié par un couple $K_p util$ - $K_s util$ de clés publique et privée RSA. Un certificat est de la forme :

$$\{K_p util, K_p inode, expiration, \mathrm{Sig}(1{+}2{+}3, K_s inode)\}$$

et donne à l'utilisateur représenté par $K_p util$ le droit d'écrire sur le fichier représenté par la clé publique $K_p inode$. Le certificat est signé avec la clé privée $K_s inode$ de l'inode, et il est valable jusqu'à la date *expiration*. Ainsi la clé privée de l'inode n'a plus à être distribuée aux différents utilisateurs autorisés à écrire sur le fichier, et de plus le droit d'écriture sur un fichier peut être retiré à un utilisateur par simple non renouvellement de son certificat.

Cette solution nécessite de modifier les contrôles d' intégrité effectués lors de l'enregistrement et de la relecture d'un inode de la façon suivante. Lorsqu'un utilisateur modifie un inode, il le signe désormais avec sa propre clé privée $K_s util$, et fournit à Past lors de l'enregistrement de l'inode le certificat prouvant qu'il est autorisé à écrire dans le fichier. Chaque noeud Past vérifie alors l'authenticité du certificat à l'aide de la clé de stockage fournie par l'utilisateur, qui est la clé publique $K_p inode$, puis vérifie l'authenticité de l'inode en vérifiant sa signature à l'aide de la clé publique $K_p util$ trouvée dans le certificat. Ces mêmes contrôles doivent être effectués par un utilisateur lors de la lecture de l'inode. Pour les faciliter, les nœuds Past stockent avec l'inode le certificat du dernier utilisateur à l'avoir mis

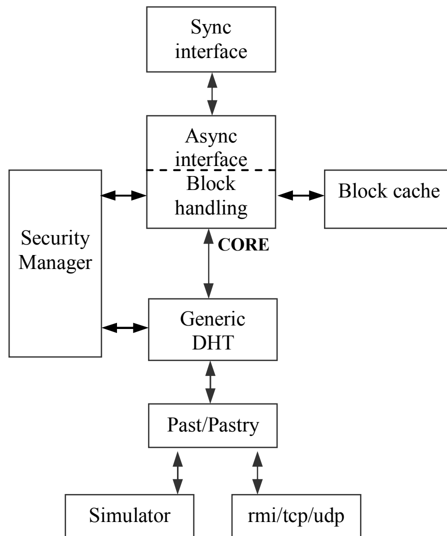

Fig. 2 – *Architecture du prototype*

à jour, et retournent à chaque lecture ce certificat en même temps que l'inode.

3.6 Confidentialité

Dans sa version actuelle, la confidentialité des données n'est pas assurée par Pastis même, qui adopte sur ce point la même politique qu'Ivy : les applications qui requièrent la confidentialité de leur données doivent les encrypter avant de les insérer dans Pastis. Des travaux sont en cours pour intégrer dans Pastis la prise en charge de la confidentialité, notamment en cas de partage du fichier.

4 Prototype

Nous avons développé en Java 1.4 un prototype de Pastis s'appuyant sur FreePastry 1.3, la version open-source de Past et Pastry. La figure 2 montre les différents composants logiciels du prototype, qui s'exécutent tous au sein d'une même JVM sur un nœud donné.

Au niveau le plus haut, Pastis offre aux applications une interface propriétaire, inspirée des classes Java *File* et *RandomAccessFile*. A la différence d'une interface NFS, celle-ci permet de clairement identifier les appels `open()` et `close()`, et ainsi de correctement gérer les différents modèles de cohérence. Au niveau le plus bas, Pastis s'interface indifféremment avec deux couches de communication, correspondant à deux modes de test.

Le premier mode consiste à faire communiquer par sockets UDP et TCP des nœuds Pastry réels situés sur des calculateurs distincts. Les délais réseau rencontrés dans un environnement large échelle sont dans ce cas émulés en utilisant un routeur DummyNet [4] intercalé entre les différents calculateurs. Le second mode de test consiste à lancer plusieurs nœuds Pastry virtuels sur un même calculateur, en les faisant communiquer par l'intermédiaire d'un simulateur de réseau. Ce simulateur permet d'introduire des délais de transmission variables entre les nœuds virtuels. Dans les deux modes de test, le code Pastry / Past / Pastis exécuté est le même, seule change la couche de communication.

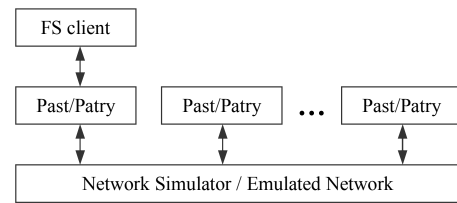

Fig. 3 – *Environnement de test*

5 Mesures

De façon à évaluer les performances de notre prototype, nous avons mis en œuvre le benchmark Andrew [6], très largement utilisé. Ce benchmark consiste à recopier une arborescence de développement et à générer des exécutables par la commmande `make`. Il comprend cinq phases : (1) création des répertoires cible, (2) recopie des fichiers source, (3) lecture des attributs des fichiers, (4) lecture de fichiers, et (5) exécution de la commande `make`. Dans le cadre des tests effectués, le répertoire de référence utilisé pour le benchmark comprend 55 fichiers source à plat, d'une taille totale de 450 Ko.

De fait, Pastis possédant pas d'interface NFS, nous avons dû développer en Java un programme qui simule ce benchmark en faisant appel à l'interface propriétaire de Pastis. Une option de lancement de ce programme nous permet de rediriger sur un répertoire de montage NFS les lectures - écritures effectuées. Ainsi, il nous est possible de comparer les performances de Pastis avec celles de NFS.

Dans tous les tests, nous exécutons une seule instance du benchmark Andrew, qui solicite le client Pastis local, lequel contacte les différents nœuds Past - Pastry pour lire et écrire les blocs de données, comme le monte la figure 3.

Phase	Simulated	WIRE
1	-	-
2	82,8	100,6
3	14,6	13,2
4	14,5	13,2
5	215,2	237,6
Total	**327,1**	**364,6**

FIG. 4 – *Simulation et émulation*

Phase	CTO	RYW	RYW
1	-	-	-
2	72,2	71,1	71,5
3	22,8	9,2	9,9
4	23,2	9,1	9,9
5	328,1	164,5	168,5
Total	**446,3**	**253,9**	**259,8**

FIG. 5 – *Modèles de cohérence*

Phase	NFS	Pastis
1	-	-
2	19,6	39,6
3	3,5	3,9
4	9,9	4,2
5	53,6	75,4
Total	**86,6**	**123,1**

FIG. 6 – *Comparaison avec NFS*

5.1 Simulation et émulation

La première série de tests menée est destinée à valider notre simulateur réseau en comparant, à conditions identiques, les résultats obtenus en environnement simulé et en environnement émulé. Dans les deux cas, la configuration utilisée comprend quatre nœuds, la latence réseau entre chaque nœud est de 100 ms, la réplication Past est désactivée, et la taille du cache local de Pastis est fixée à 2 Mo, ce qui est suffisant pour contenir l'ensemble des fichiers, y compris les exécutables générés.

La figure 4 présente les résultat de ce test pour chaque phase du benchmark, et montre que l'exécution en environnement simulée est environ 10% plus rapide qu'en environnement émulé. Ceci peut s'expliquer par le fait que dans l'environnement émulé, les messages échangés subissent un processus de sérialisation et désérialisation Java et traversent les différentes couches réseau, ce qui n'est pas le cas dans l'environnement simulé.

Dans la suite, nous supposons que l'écart constaté entre les deux modes de mesure reste le même, malgré l'augmentation du nombre de nœuds et l'activation de la fonction de réplication Past. Cette hypothèse, qui semble raisonnable, doit néanmoins être confirmée par des mesures plus approfondies.

5.2 Modèles de cohérence

La deuxième série de tests a pour but de comparer les performances des deux modèles de cohérence implémentés. On utilise pour cela une configuration simulée comprenant 4096 nœuds, la latence réseau entre deux nœuds étant de 50 ms, et Past étant configuré pour générer 16 répliques par bloc inséré.

La figure 5 montre pour chaque phase du benchmark le résultat des tests sur trois exécutions : la première avec le modèle close-to-open, la seconde avec le modèle read-your-writes, et la troisième avec ce même modèle, mais en faisant en sorte que 10% des nœuds ne possèdent pas la version de l'inode cherchée, suite par exemple à une panne les ayant empêchés d'être mis à jour. Comme prévu, les résultats montrent que le modèle read-your-writes apporte un gain très significatif par rapport au modèle close-to-open. Ce gain est de 43% toutes phases confondues dans des conditions favorables, et reste de 42% en présence de 10% de pannes ou reconfigurations réseau.

5.3 Comparaison avec NFS

La dernière série de tests vise à comparer Pastis à NFS. On utilise pour Pastis une configuration simulée comprenant 128 nœuds, avec une latence réseau entre deux nœuds de 25 ms, et un facteur de réplication Past de 16. La configuration NFS est composée d'un client et d'un serveur NFS, la latence réseau pour un aller-retour RPC entre le client et le serveur étant de 50 ms.

La figure 6 montre le résultat de cette comparaison pour chaque phase du benchmark. Toutes phases confondues, Pastis est environ 43% plus lent que NFS. Des mesures complémentaires suggèrent que passer de 128 à 4096 nœuds sur la configuration Pastis accroît le temps d'exécution total d'environ 40%, ce qui rendrait Pastis à peine 100% (2 fois) plus lent que NFS dans cette nouvelle configuration.

6 Conclusion et travaux futurs

Nous avons implémenté un système de fichiers pair à pair multi-utilisateur dont l'architecture permet de supporter un très grand nombre d'écrivains. Dans un environnement où les latences réseau sont très élevées, l'utilisation de Pastry et Past comme support de stockage permet d'atteindre de bonnes performances grâce à leurs propriétés de localité. Nous avons encore amélioré ces peformances par la définition et la mise en œuvre de modèles de cohérences relachés appropriés, permettant de tirer parti de toutes les caractéristiques du réseau pair à pair sous-jacent. Les premières mesures de performances, qui doivent être confirmées, suggèrent que Pastis est à peine deux fois plus lent que NFS à configurations comparables.

Pastis fait l'objet de plusieurs travaux en cours, répartis sur trois axes. Il s'agit d'abord de développer de nouveaux modèles de cohérence pour couvrir une large gamme de besoins applicatifs, et permettre notamment la sérialisation des mises à jour. Il faut également améliorer la gestion de la sécurité pour prendre en compte de façon plus efficace les changements de droit d'accès et la protection de la confidentialité des données. Il s'agit enfin de développer un mécanisme de ramasse-miette distribué permettant de détruire les CHB qui ne sont plus utilisés suite aux mises à jour.

Références

[1] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA. ACM*, June 1997.

[2] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[3] I. Clarke, O. Sandberg, B.Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46-66, July 2000.

[4] L. Rizzo. Dummynet and Forward Error Correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, June 1998.

[5] FreePastry. `http://freepastry.cs.rice.edu/`

[6] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, volume 6, February 1988.

[7] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[9] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Cambridge, MA, November 2000.

[10] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. In *ACM Computing Surveys*, 22(4):321-375, Dec. 1990.

[11] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation* (OSDI 2002).

[12] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation* (OSDI 1999).

[13] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, August 2001.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (SOSP '01), Chateau Lake Louise, Banff, Canada, Oct. 2001.

[15] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *5th Symp. on Op. Sys. Design and Implementation (OSDI)*, Boston, MA, USA, December 2002.