

Le Network File System de Sun (NFS)

- [Le Network File System de Sun \(NFS\)](#)
 - [Architecture](#)
 - [Protocoles](#)
 - [Mounting](#)
 - [Automounting vs Static mounting](#)
 - [Directory et accès aux fichiers](#)
 - [Problèmes](#)
 - [Implémentation](#)
 - [MOUNT](#)
 - [OPEN](#)
 - [READ](#)
 - [WRITE](#)
 - [Cohérence du cache](#)
-

Network File System de Sun (NFS)

NFS = système de fichiers distribué.

Idée de base = pouvoir partager entre plusieurs clients et serveurs hétérogènes un **même** système de fichiers.

Une machine peut être à la fois client et serveur.

Un serveur NFS **exporte** ses directories pour qu'elles soient accessibles par des clients.

Si une directory est exportée, c'est tout le sous-arbre qui est exporté.

Liste des directories exportées dans `/etc/exports`.

Architecture

Un client qui veut accéder à une directory distante doit la **monter** dans sa propre hiérarchie.

Une station cliente sans disque (diskless) peut faire "comme si" elle avait un disque en montant des systèmes distants.

Une station avec un disque local aura une hiérarchie en partie locale et distante.

Pour les programmes du client ➡ pas de différence entre fichiers locaux ou distants.

Si deux clients ont monté la même directory, ils en partagent les fichiers.

➡ simplicité de NFS.

Protocoles

NFS doit supporter des systèmes hétérogènes (clients DOS utilisant des processeurs Intel, serveurs tournant sur Sun Sparc, ...).

↳ clients et serveurs utilisant différents OS et différentes machines.

Il est impératif de définir une bonne interface client/serveur.

Avantage d'une interface clairement définie : possibilité d'écrire de nouveaux clients et serveurs compatibles.

2 protocoles sont définis.

1 protocole pour le **mounting** et 1 protocole pour la **directory** et l' **accès aux fichiers**.

Mounting

Soit *C* le client et *S* le serveur.

C envoie à *S* un chemin d'accès (le nom de la directory à monter) et demande la permission de monter la directory chez lui.

L'endroit où *C* va monter la directory n'est pas important pour *S*.

Si le chemin d'accès est correct et si la directory se trouve dans `/etc/exports`, *S* renvoie un **file handle** à *C*.

Le **handle** est composé :

- du type du système de fichiers ;
- du disque ;
- du numéro de i-node de la directory ;
- d'infos de sécurité (droits d'accès).

Pour lire ou écrire dans la directory montée, il faut utiliser ce handle.

Un client peut monter des directory sans intervention humaine.

Ces clients ont un fichier `/etc/rc` shell script qui contient les commandes de mount et lancé automatiquement au boot.

C'est le **static mounting**.

Les versions récentes de Sun Unix ont l' **automounting** :

Des directory distantes sont **associées** à des directories locales, mais elles ne sont pas montées, et leurs serveurs ne sont pas contactés au boot.

La première fois qu'un client accède à un fichier distant, les serveurs sont contactés. Le premier qui répond gagne.

Automounting vs Static mounting

Avantages de l'automounting sur le static mounting :

1. si un des serveurs NFS nommé dans `/etc/rc` est down ↳ difficile de mettre en route le client ;
2. dans le static mounting, on ne contacte qu'un serveur pour chaque directory, alors qu'on peut en contacter plusieurs dans le automounting ↳ tolérance aux fautes.

Inconvénient : tous les serveurs "alternatifs" pour une même directory doivent être cohérents ↪ surtout utilisé pour des systèmes de fichiers **read-only**.

Directory et accès aux fichiers

2ème protocole.

Les clients envoient des messages pour manipuler des directories, lire et écrire des fichiers et leurs attributs (taille, date de modification, propriétaire, etc.).

Tous les appels systèmes sont pris en charge par NFS sauf OPEN et CLOSE.

OPEN et CLOSE ne sont pas utiles :

- pour chaque opération read ou write, le client d'abord envoie une demande LOOKUP qui renvoie un file handle, **le serveur ne garde pas trace de cette demande** ;
- une opération read ou write est accompagné du handle.

Si le serveur crashe ↪ aucune info sur les fichiers ouvert est perdue (puisqu'il n'y en a pas).

Un serveur NFS est **stateless**.

Problèmes

Un fichier Unix peut être ouvert et verrouillé (locked) pour empêcher les autres processus de l'utiliser.

Fichier fermé ↪ verrous relâchés.

NFS est stateless, on ne peut pas associer de verrous à l'ouverture d'un fichier.

Il faut un mécanisme **externe** à NFS pour gérer le verouillage.

NFS utilise quand même le système de protection Unix (bits `rwX` pour le owner, group et world).

MAIS : le serveur NFS **croit toujours** le client pour valider un accès.

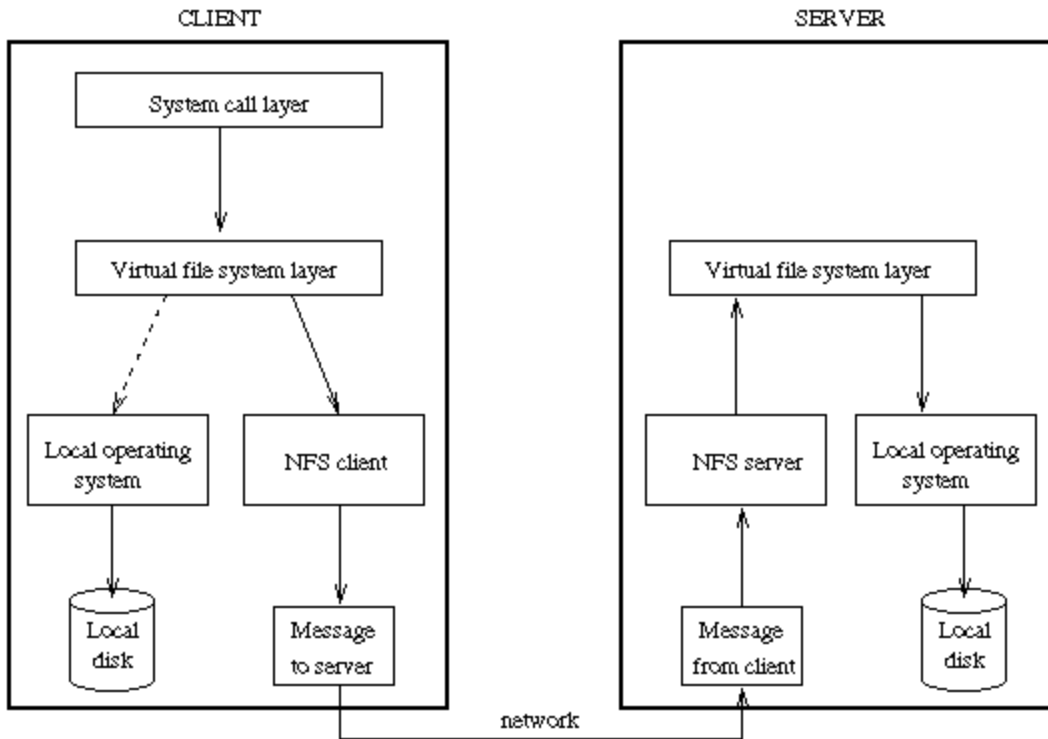
Que faire si le client ment ?

Utilisation de la cryptographie pour valider les requêtes.

Problème : les données, elles, ne sont pas cryptées.

Les clés sont gérées par le NIS (**Network Information Service**, ou **yellow pages**)

Implémentation



La couche VFS maintient une table pour chaque fichier ouvert.

Chaque entrée est un *v-node* (virtual i-node). On indique si le fichier est local ou distant.

Exemple : la séquence < MOUNT, OPEN, READ >.

MOUNT

MOUNT :

- le *sysop* envoie *mount* + remote directory + local directory + other ;
- le programme *mount* parcourt le nom de la remote dir et trouve le nom de la machine distante associée ;
- *mount* contacte la machine et demande un handle pour cette directory ;
- le serveur renvoie le handle si la requête est correcte ;
- *mount* fait un appel système MOUNT (kernel).

Le kernel a la main :

- il construit un **v-node** pour la remote dir ;
- demande au client NFS de créer un **r-node** (remote i-node) dans sa table pour le file handle ;
- le v-node pointe sur le r-node.

OPEN

OPEN :

- le kernel parcourt le nom du chemin d'accès, trouve la directory, voit qu'elle est distante et dans le v-node de la directory trouve le pointeur sur le r-node ;
- le kernel demande au client NFS d'ouvrir le fichier ;
- le client NFS récupère le nom du serveur dans le nom du chemin d'accès et un handle ;
- le client crée un r-node et averti la VFS qui crée un v-node pointant sur le r-node ;

- le processus appelant récupère un file descriptor, relié au v-node de la VFS.

Côté serveur, **rien n'est créé**.

READ

READ :

- la VFS trouve le v-node correspondant ;
- la VFS détermine si c'est local ou distant et quel est le i-node ou r-node à utiliser ;
- le client NFS envoie une commande READ, avec le handle + l'offset.

Les transferts se font normalement 8ko / 8ko, même si moins d'octets sont demandés.

Automatiquement, dès que le client a reçu les 8ko demandés, une nouvelle requête de 8ko est envoyée.

C'est le **read ahead**.

WRITE

Les transferts se font aussi 8ko / 8ko.

Tant que les données écrites sont < 8ko, elles sont accumulées localement.

Dès que le client a écrit 8ko, les 8ko sont envoyé au serveur.

Quand un fichier est fermé, ce qui reste à écrire est envoyé au serveur.

Utilisation du **caching** :

les clients ont 2 caches : attributs et données.

⇒ problèmes de **cohérences**.

Cohérence du cache

Pas de solution "propre" : on essaie de réduire le risque au maximum, mais sans l'éviter tout à fait.

- Un timer est associé à chaque entrée du cache. Quand le timer expire, l'entrée est annulée. Normalement 3s pour les données et 30s pour les attributs.
- Quand un fichier "caché" est ouvert, le serveur est contacté pour savoir la date de la dernière mise-à-jour. Si MAJ plus récente que la copie, l'entrée est annulée.
- Chaque 30s un timer expire et toutes les entrées sales sont envoyées au serveur.



[Retour au sommaire.](#)