9P, LE PROTOCOLE PARFAIT POUR L'IOT

Édouard KLEIN

hedk@beaver-labs.com

Plutôt que de faire passer du JSON sur la couche HTTP, faisons un saut dans le passé pour voir à quoi le futur pourrait ressembler. Développé à partir de la fin des années 80 à Bell Labs, le système d'exploitation Plan 9 utilise le protocole 9P pour rendre le réseau cohérent, sûr et transparent, trois caractéristiques qui manquent à l'IoT...

mots-clés : PROTOCOLE / PLAN 9 / IOT / SÉCURITÉ / RÉSEAU

Les laboratoires Bell ont changé la face du monde avec le transistor, les cellules photovoltaïques, la théorie de l'information, le C, Awk, UNIX et bien d'autres.

Étudions aujourd'hui le système d'exploitation Plan 9 from Bell Labs dont les principaux développeurs d'UNIX ont contribué.

Il s'est affranchi de la compatibilité POSIX et présente une interface de programmation (*Application Programming Interface*, API) épurée et cohérente dans un contexte distribué : si UNIX a réglé le problème de coordonner plusieurs utilisateurs sur une machine (*Time Sharing*), Plan 9 permet de coordonner plusieurs utilisateurs sur plusieurs machines, distribuées dans un réseau potentiellement hostile.

Le protocole 9P sur lequel se base cette API peut, à travers son port sur Linux et des implémentations sur le métal, nous aider à bâtir un IoT fiable et agile.

1. SOUS UNIX NAQUIT LE FICHIER

1.1 Les opérations élémentaires

L'une des innovations majeures d'UNIX, à la fin des années 60, est le système de fichiers qui associe des données arbitraires à un nom hiérarchique.

Avant de pouvoir manipuler ces données, un processus doit acquérir la ressource en demandant l'ouverture (appel système `open`) du fichier. Le noyau vérifie la disponibilité de la ressource et les autorisations du demandeur.

Si le test est concluant, le noyau fournit au processus un descripteur de fichier, un simple entier qui est une référence vers le fichier ouvert, valable tant que le fichier ne sera pas fermé (appel système `close`).

Armé de ce descripteur, le processus peut (si le fichier a été ouvert avec le mode adéquat) lire les données du fichier (appel système `read`) ou les modifier (appel système `write`).

Il existe d'autres opérations annexes, comme par exemple des opérations permettant de modifier les permissions.

1.2 Sous Plan 9, tout est fichier. Tout ? Oui, tout.

Plan 9 étend cette API simple, mais générique à tout le système d'exploitation.

Les fichiers sous Plan 9 sont aussi une interface permettant à un processus de communiquer avec un autre, ou avec le noyau.

Ainsi, la plupart des fonctionnalités de Plan 9 sont offertes à travers des programmes (les serveurs) qui créent des fichiers synthétiques, sont destinataires des opérations habituelles (`open`, `read`, `write`, `close`) sur ces fichiers et réagissent dynamiquement à ces opérations.

Le programme qui a recours à ces appels système (le client) ne voit aucune différence entre ouvrir un fichier sur le disque ou parler à un programme.

Exit donc les sockets ! Pour communiquer via le réseau sous Plan 9, l'on manipule des fichiers dans `/net` **[slashnet]**. En lisant et en écrivant dans le fichier `/net/tct/2/data`, l'on fait la même chose qu'en utilisant les fonctions `recv` et `send` de l'API des sockets sous UNIX.

Exit également le serveur X ! L'écran est le fichier `/dev/screen`, la souris `/dev/mouse` et le clavier `/dev/kbd`. Lire ce dernier retournera ce que l'utilisateur tape.

Un résultat immédiat est une réduction drastique du nombre d'appels système : 38 aujourd'hui contre quelques centaines pour Linux, BSD ou Windows.

1.3 /dev/door, /dev/thermostat

Dans le meilleur des cas aujourd'hui, un dispositif connecté fournit une interface HTTP(S) avec laquelle on doit communiquer en JSON ou XML. Bien souvent, l'interface se dit RESTful et oblige le client à sauvegarder l'état de la connexion et à le renvoyer au serveur à chaque requête.

C'est loin d'être idéal, mais c'est moins pire qu'une API dans un ou deux langages de programmation exclusivement ou, désastre, un blob binaire propriétaire.

La leçon à tirer de Plan 9 est qu'il suffit de présenter un système de fichiers synthétique pour avoir une interface claire et accessible à tous les langages. Par exemple, une porte sera représentée par un seul fichier, dans lequel l'on pourrait lire ou écrire les mots `open` et `close`. Un thermostat serait un dossier avec un fichier par pièce, et l'on pourrait y lire la température courante, et y écrire la température désirée. Une enceinte Bluetooth pourrait être un fichier où l'on écrit le .wav du son que l'on veut émettre, etc.

2. PLAN 9 APPORTA LA TRANSPARENCE RÉSEAU

2.1 Le protocole 9P

Plan 9 rend de plus ces opérations sur les fichiers virtuels complètement équivalentes que le serveur et le client soient sur une même machine ou séparés par un océan !

Puisque les fichiers ne sont plus des suites de bits sur un disque, mais un moyen de communiquer avec un programme qui réagira aux appels à `open`, `read`, `write`, `close`, etc., il faut pouvoir transcrire ces appels et les faire parvenir au programme distant.

Le protocole 9P **[man9p]** répond à ce besoin en transformant chaque appel à une fonction de manipulation de fichier en un message. Chaque message est composé :

- d'un type correspondant à l'opération concernée, par exemple un appel à `write` génèrera un ou plusieurs messages de type `Twrite`, auxquels le serveur répondra par un ou plusieurs `Rwrite` ;
- d'un tag (un simple entier) permettant de savoir à quelle requête (messages dont le type commence par T) correspond une réponse (dont le type commence par R) ;
- d'une charge utile qui varie en fonction du type. Par exemple, un `Twrite` contient :
  - un descripteur du fichier ouvert où il faut écrire ;
  - l'index (en octets) à partir duquel écrire ;
  - le nombre d'octets à écrire ;
  - les données à écrire.

Le `Rwrite` contient uniquement le nombre d'octets qui ont effectivement été écrits.

Puisque ce protocole permet sans perte de généralité de transformer n'importe quelle série d'appels à `open`, `read`, `write`, `close`, etc., en une suite de messages, il n'est pas utilisé que sur le réseau, mais aussi par le noyau Plan 9 pour faire communiquer deux processus sur une même machine.

Les serveurs, sous Plan 9, sont d'une simplicité extrême, puisqu'ils se contentent de lire du 9P sur leur entrée standard et d'afficher la réponse sur leur sortie standard.

Le *démultiplexing* des connexions et la sécurité sont assurés par des programmes spécialisés, communs à tous les serveurs.

TCP est bien entendu en mesure de porter du 9P, mais 9P se propage sur n'importe quel protocole ordonné bidirectionnel sans perte. Il était originellement transmis avec IL, et a déjà été déployé avec succès via un lien infrarouge **[styxbrick]**.

2.2 IoF, the Internet of Files

Il existe plus d'une quarantaine d'implémentations de 9P **[9ps]**. Certaines peuvent être embarquées directement sur le métal. Un protocole comme PJON **[pjon]** permet d'éviter la complexité de la pile TCP tout en fournissant le support ordonné bidirectionnel sans perte dont a besoin 9P.

Comme le noyau Linux comprend le protocole 9P grâce à v9fs **[v9fs]**, avec une simple commande :

```
mount -t 9p 10.10.1.2 /mnt/radiateur
```

un serveur 9P écoutant sur une machine distante peut apparaître comme un dossier contenant des fichiers synthétiques sur la machine locale.

Ce serveur peut être une autre machine Linux qui, avec une simple redirection de l'entrée/sortie standard de **9pserve** **[9pserve]** rend disponible un dispositif branché à travers une liaison série. Le dispositif lui-même ne possède pas d'OS, mais parle 9P sur le port série, ce qui se fait en quelques dizaines de lignes de C.

Il est ainsi possible, à coups de **mount**, de se bâtir une vision locale et cohérente d'un système hétérogène et distribué : imaginons un bâtiment intelligent où chaque pièce est bardée de capteurs et d'actuateurs. 9P permet aux différents capteurs et actuateurs de rester extrêmement simples, et donc fiables et peu onéreux. Ils n'ont pas besoin de systèmes d'exploitation complexes et présentent leurs fonctionnalités via 9P à travers une liaison physique (câble série, radio faible puissance, etc.). Ces liaisons convergent pour chaque pièce, ou chaque étage, vers un micro ordinateur comme par exemple un Raspberry Pi, qui redirige les messages 9P vers l'entrée sortie standard de **9pserve**.

N'importe quel client peut maintenant, avec un appel à **mount -t 9p**, se connecter aux instances de **9pserve** qui l'intéressent et par là faire parvenir des messages aux dispositifs physiques.

Il est possible de bâtir, pour chaque fonction du système, une vue spécifique. Ainsi, le programme régulant la température fonctionnera dans un dossier où seront seuls montés les capteurs de température, et les actuateurs des radiateurs et des fenêtres, tandis que l'éclairage fonctionnera dans un dossier contenant les points de montage des stores et des éclairages. Un dysfonctionnement ou une compromission d'un des deux systèmes de régulation ne pourra pas donc pas avoir d'impact sur l'autre puisque les actuateurs de l'un sont invisibles à l'autre.

3. 9P ET SÉCURITÉ

Dans notre exemple, les capteurs et actuateurs n'ont pas besoin d'être sécurisés, car la nature de leur liaison physique fait qu'un attaquant y ayant accès se trouve nécessairement dans la pièce concernée, et pourrait tout aussi bien ouvrir la fenêtre à la main.

En revanche, l'accès distant via le Raspberry Pi se doit d'être contrôlé. 9P permet ce contrôle d'une manière fine et intuitive.

3.1 Permissions

L'accès aux fichiers UNIX est défini par quatre capacités (lecture, écriture, exécution de fichier et traversée d'un répertoire) attribuées à trois classes d'utilisateurs (le propriétaire, un groupe, et le reste du monde) **[chmod]**.

C'est grâce à l'appel système **open**, préalable à toute manipulation de fichier, que le noyau vérifie que les capacités accordées par les permissions du fichier au propriétaire du processus lui permettent d'effectuer l'ouverture selon le mode (lecture seule, lecture-écriture, etc.) qu'il demande.

Cette gestion des droits est également étendue aux systèmes de fichiers virtuels montés avec la commande **mount -t 9p**. Ainsi, en montant les capteurs et actuateurs d'un étage dans des dossiers appartenant à divers utilisateurs du Linux tournant sur le Raspberry Pi de l'étage, l'on peut se décharger du contrôle d'accès sur le noyau, qui l'implémente de manière fiable une fois pour toutes.

Avant de pouvoir accéder à un capteur ou actuateur de manière distante, il faudra prouver son identité au noyau du Raspberry Pi, via SSH par exemple.

3.2 Authentification

Le principal défaut d'UNIX en termes de sécurité ne concerne pas les permissions des fichiers, mais la gestion du propriétaire d'un processus qui est héritée du processus parent, ou bien copié depuis le propriétaire du fichier contenant le programme.

Une fois lancé, un processus ne peut changer d'identité que si son propriétaire est root. Cela oblige les processus devant changer d'identité au cours de leur vie à démarrer comme utilisateur privilégié, ce qui est un trou béant de sécurité que les contre-mesures introduites au fil des ans (*pledge*

sous OpenBSD, *capabilities* sous Linux, etc.) peinent à combler.

De plus, l'identité est une propriété locale, manipulée par le noyau d'une seule machine, alors que 9P est un protocole réseau, mettant à disposition sous l'aspect de fichiers des ressources appartenant à des machines qui ne connaissent peut-être pas sous le même nom un même utilisateur.

Les remédiations modernes à ce problème (Kerberos, distribution de clefs SSH, Active Directory) sont des monstres de complexité.

À l'inverse, la solution adoptée par Plan 9 est simple et élégante **[auth]**. Chaque agent et chaque machine disposent d'un dépositaire de secrets appelé factotum, sécurisé comme peut l'être ssh-agent sur un UNIX. Lors de l'établissement d'une connexion 9P, le factotum du client et celui du serveur discutent ensemble via un descripteur de fichier particulier, jusqu'à ce que chacun soit convaincu de l'identité de l'autre.

Une fois cette conversation accomplie, le descripteur de fichier est maintenant un token d'authentification permettant au client de monter le système de fichiers fourni par le serveur.

Un port expérimental de ce mécanisme d'authentification sous Linux a montré des résultats encourageants **[vrs]**, mais en attendant sa fiabilisation, SSH reste la meilleure solution.

3.3 Les descripteurs sont des tokens

Il existe enfin un dernier mécanisme de sécurité inhérent à la construction du protocole 9P en tant qu'abstraction réseau des opérations sur les fichiers.

À la différence du descripteur de fichier renvoyé par open sous UNIX qui est un entier valable uniquement dans le contexte d'un processus donné, le descripteur renvoyé par un message 9P `Ropen` est valide non seulement pour tous les processus du système, mais aussi pour des processus distants.

Il est ainsi possible d'ouvrir en lecture seule un fichier pour lequel on a pourtant aussi les droits en écriture, et de fournir le descripteur à un autre processus, sur une autre machine, pour qu'il puisse à l'aide de messages 9P `Tread`, en lire le contenu sans avoir besoin de s'authentifier et sans pouvoir y écrire. Ce token se répudie simplement en fermant le fichier (message `Tclose`).

CONCLUSION

9P permet de bâtir un IoT :

- où chaque serveur, plutôt que d'intégrer des mécanismes de sécurité lacunaires et défaillants, se base sur la gestion faite par le noyau Linux de l'identité des utilisateurs et des permissions de fichiers pour assurer le contrôle d'accès ;
- où chaque client se compose une vue spécifique du système en montant où il le souhaite les services dont il a besoin, et y accède via une interface universelle, sécurisée, simple et facile à débugger que sont les appels systèmes `open`, `read`, `write`, `close`, etc. ;
- où chaque client ayant accès à une ressource peut en autoriser l'accès (potentiellement en le restreignant au passage) à d'autres clients et révoquer cet accès en un seul message ;
- où en forçant le serveur à une gestion minime de l'état d'une connexion, l'on simplifie énormément le code client, à la différence des API REST, qui poussent toute la gestion de l'état dans le client. ■

RÉFÉRENCES

[slashnet] The Organization of Networks in Plan 9, Dave Presotto, Phil Winterbottom : http://doc.cat-v.org/plan_9/4th_edition/papers/net/

[man9p] man 9p intro : https://9fans.github.io/plan9port/man/man9/intro.html

[styxbrick] Styx on a brick, Chris Locke : http://doc.cat-v.org/inferno/4th_edition/styx-on-a-brick/

[9ps] Liste des implémentations de 9P : http://9p.cat-v.org/implementations

[pjon] Protocole PJON pour l'IoT : https://www.pjon.org/

[v9fs] v9fs: Plan 9 Resource Sharing for Linux : https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/Documentation/filesystems/9p.txt

[9pserve] man 9pserve : https://9fans.github.io/plan9port/man/man4/9pserve.html

[chmod] man chmod

[auth] Security in Plan 9, Cox et al. : http://doc.cat-v.org/plan_9/4th_edition/papers/auth

[vrs] Dr Glendarme, Klein et Gette : http://beaver-labs.com/9pvrs