


Rapport projet LO14

Script Vsh



I ❤️ #!/bin/bash

21 juin 2014

Julien Grave
SRT1

Lucas Soulier
SRT1

Table des matières

1	Introduction	3
2	Méthodes employées	4
2.1	Architecture de transmission	4
2.2	Un seul maître à bord	4
3	Fonctionnement technique	5
3.1	Les modes list et extract	5
3.2	Le mode browse	5
3.2.1	Les commandes ls, cd et cat	5
3.2.2	La commande rm	6
3.3	Les sécurités	6
3.4	Schéma d'exécution d'une requête	6
4	Conclusion	7

1 Introduction

Vsh est un script bash dont le but est de proposer un serveur d'archivage mais également le système pour s'y connecter et pouvoir naviguer et/ou extraire une archive depuis un hôte distant.

Dans le cadre de la réalisation de ce projet, il nous est apparu évident dans un premier temps de scinder notre projet en Client/Serveur.

- Côté client : nous gérons l'envoi et la réception de requête ;
- Côté serveur : nous traitons les requêtes client.

L'objectif que nous nous sommes fixé ensuite est de réaliser l'entièreté du script en *Bash*, c'est pourquoi nous n'employons pas *awk*. Tout ceci, afin de réellement progresser dans la compréhension du Bash et plus particulièrement dans l'utilisation des « regex ». En outre, nous avons pris l'initiative de mettre en place dès le début des conventions de nommage entre nous et surtout de rendre notre code le plus souple possible en codant des fonctions réutilisables. Ce qui a également l'avantage de factoriser grandement le code. Nous nous sommes basés sur les conventions disponibles ici : `Bash scripting conventions`.

Enfin, nous avons travaillé de manière collaborative en utilisant Git. Vous pouvez d'ailleurs consulter notre code source à l'adresse suivante : <https://github.com/lonk/lo14>

2 Méthodes employées

2.1 Architecture de transmission

Afin de réaliser la transmission client/serveur, nous nous sommes dirigés tout naturellement vers *Netcat*, commande vue en TP et très simple d'utilisation. Néanmoins, nous avons rapidement fait face à des problèmes de réception des messages avec la version OpenBSD. De même, nous n'avons pas choisi la version traditionnelle car elle ne peut maintenir son écoute côté serveur que par le biais d'une boucle « infinie », ce que nous estimons ne pas être très « propre » en terme de programmation.

C'est pour cette raison que nous nous avons finalement choisi *Ncat*, version Netcat remastérisée par les développeurs de Nmap qui est beaucoup plus stable et qui par ailleurs supporte les connexions multiples par le biais de l'option -e qui exécutera un nouveau processus du script à chaque nouvelle connexion client. Ce qui donne un Netcat vraiment efficace.

En pratique, nous avons d'un côté notre serveur qui reste à l'écoute des connexions et de l'autre nos clients qui peuvent envoyer une requête à n'importe quel moment, sans que cela ne crée de conflit entre eux grâce au support de connexions multiples.

2.2 Un seul maître à bord

Tout ce qui est demandé par le client est au maximum exécuté côté serveur pour protéger l'intégrité du fonctionnement de notre système. Il est évident que l'effet négatif d'un tel fonctionnement est que notre serveur met un laps de temps à répondre aux requêtes. C'est un choix assumé qui reste pour l'expérience utilisateur tout à fait supportable. En moyenne, nous avons observé un temps de réponse de moins d'une seconde !

3 Fonctionnement technique

3.1 Les modes *list* et *extract*

Il n'est pas utile de décrire dans les détails le mode *list* qui dans les faits a plus un aspect pratique que complexe.

Concernant le mode *extract*, il est le seul dont le code soit côté client puisqu'il agit sur le système de ce dernier. Dans un premier temps, les répertoires et fichiers sont créés après « passage » de l'architecture puis dans un second temps les droits sont répercutés. Il est très important d'appliquer les droits après la création des répertoires et fichiers pour éviter des problèmes de création de répertoires ou fichiers dû aux droits du répertoire parent.

3.2 Le mode *browse*

3.2.1 Les commandes *ls*, *cd* et *cat*

Ces trois commandes ont été les plus simples à réaliser. Néanmoins, nous avons pris le soin de les développer de manière consciencieuse pour que les fonctions soit totalement réutilisables pour créer d'autres commandes plus complexes. Parmi les fonctions les plus essentielles, il y a `get_full_path`. Cette fonction permet de traduire n'importe quel chemin indiqué par le client en chemin complet.

Exemple : si le client est à la racine et qu'il précise comme chemin `A/A1/../../B` la fonction retournera `Exemple/Test/B` (dans le cas de l'architecture exemple du sujet). L'enchaînement de cette fonction est le suivant :

1. On traduit le chemin fournit en chemin absolu si ce n'en est pas déjà un.
2. On lit ensuite le chemin de gauche à droite en interprétant tous les doubles points qu'il contient pour obtenir le chemin « simplifié ».
3. Enfin, grâce à la fonction `get_root_path` on récupère le chemin racine (`Exemple/Test` dans l'archive exemple du sujet) et on le concatène au début du chemin dit simplifié.

Finalement, grâce à cette fonction, nous pouvons traiter tout le restant de l'exécution de nos commandes sans aucun soucis puisque nous savons que le chemin traité sera exactement celui spécifié dans notre archive.

3.2.2 La commande rm

Sans doute la fonction la plus difficile à réaliser. Nous avons créé une fonction pour le mode non-récuratif mais qui soit réutilisable « en boucle » par le mode récursif, tout en prenant soin de ne pas créer de conflit entre les différentes suppressions de lignes. C'est pour cela qu'en mode récursif (rm -r), un tableau des lignes architecture à supprimer est généré, et que les suppressions ont lieu à la fin de bas en haut. C'est aussi la raison pour laquelle les recalculs des lignes de « contenu » se font à la volée, tant qu'aucune ligne n'a encore été supprimée.

3.3 Les sécurités

À chaque appel du script Vsh, chaque argument est vérifié scrupuleusement : est-ce bien une IP valide ? L'archive est-elle existante côté serveur ? etc.

Il y a bien évidemment une fonction qui s'assure de l'existence d'un chemin ou fichier demandé par l'utilisateur et qui lui notifie dans le cas où la fonction retourne une erreur.

Dans le cas d'une erreur d'exécution, nous avons employé des codes retour différents afin de distinguer les différentes erreurs et pouvoir les identifier et le notifier au client. Exemple : trop de double points, ce qui a pour effet de notifier l'utilisateur que le répertoire n'existe pas puisqu'il a saisi trop de double points.

3.4 Schéma d'exécution d'une requête

Vous trouverez en annexe page 8 un schéma explicatif qui permet de mieux cerner l'exécution de chacune des commandes côté serveur.

4 Conclusion

Nous avons pu, grâce à ce projet, considérablement améliorer nos connaissances en *Bash* en plus de celles acquises en cours et TP. Par ailleurs, Il a été très intéressant d'apprendre à utiliser *Ncat*.

Concernant les méthodes utilisées, le point négatif concernant *Ncat* est que la transmission se fait en clair. Il aurait pu être intéressant d'utiliser le paquet *Cryptcat* qui est une version de Netcat qui chiffre la transmission. Par ailleurs, il aurait aussi été meilleur de coder davantage la fonction *extract* côté serveur pour respecter le principe d'un « seul maître à bord ». Ce sont toutes deux des pistes d'amélioration possible pour le projet.

Enfin, un autre mode qui aurait été très intéressant à développer est un mode *compile* dont la fonction aurait été inverse à celle de *extract*. Ce mode aurait permis à partir d'un dossier spécifié par l'utilisateur de générer une archive comme celle fournie en exemple du sujet.

