DOCUMENTATION

I. CRÉATION DES VM

Pour ce projet, on va avoir besoin de deux vm (virtual machine), un client et un serveur. Une fois que les machines ont été installées, on va saisir quelques commandes pour commencer.

```
$ apt-get update
$ apt-get install openssh-server git vim
```

Ceci va mettre la machine mise à jour et installer les paquets nécessaires pour le fonctionnement du projet.

→ **git**
Utiliser pour récupérer le projet.

→ **openssh-server**
Utiliser pour être capable d'envoyer / récupérer des fichiers depuis le serveur.

→ **vim**
Utiliser pour éditer des fichiers (ce n'est pas nécessaire mais utile)

Maintenant que tout est installé, on va commencer à configurer quelques fichiers.

```
$ vim /etc/ssh/sshd_config
```

Dans le fichier `/etc/ssh/sshd_config` on va faire quelques modifications

→ Décommenter la ligne `#PermitRootLogin prohibit-password`
→ Et le modifier en `PermitRootLogin yes`

Ceci va nous permettre de se connecter à une machine avec ssh en saisissant le mot de passe

Redémarrer le ssh pour prendre en effet les changements.

```
$ systemctl restart ssh
```

Activation au démarrage du ssh.

```
$ systemctl enable ssh
```

```
$ vim /etc/network/interfaces
```

Dans le fichier `vim /etc/network/interfaces` on va ajouter ce code là

```
auto ens33
iface ens33 inet static
address 192.168.64.4 #chosen address
netmask 255.255.255.0
gateway 192.168.64.2 #gateway only in .2
dns-nameservers 8.8.8.8
```

Ensuite…

```
$ vim /etc/resolv.conf
```

Dans le fichier `/etc/resolv.conf` on va ajouter cette ligne là

```
nameserver 8.8.8.8
```

Pour terminer…

```
$ systemctl restart networking
$ ping google.fr
```

Si toutes les modifications ont été faites, notre machine aura une adresse ip statique. Le ping est utilisé pour vérifier le fait qu'on a accès à internet. Avoir une adresse statique est important pour le serveur puisque sinon on devra modifier le script chaque fois que le serveur change d'adresse ip. Cette étape doit être faite pour chaque machine.

II. INSTALLATION

Premièrement, il faut télécharger le dossier qui contient tous les fichiers nécessaires pour le fonctionnement de la sauvegarde.

```
$ git clone
https://ytrack.learn.ynov.com/git/mmihail/infra_sauvegarde.git
```

Ceci va cloner un dossier appelé **infra_sauvegarde**. C'est ici qu'on va trouver notre script init.sh que nous allons exécuter.

```
$ ./init.sh
```

Le script init.sh a comme de but de
→ Déplacer le dossier actuel au `/backup`
(ne pas déplacer / renommer ce dossier)
→ Ajouter les alias nécessaires dans .bashrc pour être capable d'utiliser les commandes
→ Ajouter dans crontab l'exécution automatique de la sauvegarde

Ensuite…
On va générer une clé ssh pour améliorer la sécurité du transfert de sauvegarde

```
$ ssh-keygen -t rsa
```

→ Appuyez sur **Entrée** (sauf si vous voulez changez l'emplacement des clés)
→ Saisir un **passphrase** (c'est comme un mot de passe)

Dans le répertoire `~/.ssh/` on va voir 2 clés
→ `id_rsa` (clé privée)
→ `id_rsa.pub` (clé publique)

Maintenant on va envoyer la clé publique à notre serveur

```
$ ssh-copy-id username@ip_address
```

Vous allez devoir saisir le mot de passe du serveur.

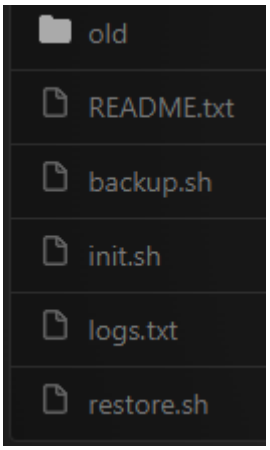
Ensuite, vous pouvez essayer de vous connecter au serveur depuis le client pour vérifier si tout fonctionne correctement.

```
$ ssh username@ip_address
```

Ceci consiste en l'installation. Maintenant vous êtes capable d'utiliser les commandes `$ backup` et `$ restore` .

III. FICHIERS - AIDE

Précédemment on a vu comment installer le système de sauvegarde. Ici on va regarder les fichiers compris dans le dossier cloné.



→ **Old/**

Un script ancien développé mais qui n'est pas utilisé puisqu'il n'est pas fonctionnel. Ce script a comme but de comparer le temps de modification (autrement appelé **mtime**) des fichiers qu'on veut sauvegarder avec le temps des fichiers sur le serveur qui a comme but d'optimiser le système de sauvegarde. De cette façon, on n'a plus besoin de sauvegarder chaque fichier à chaque fois qu'on exécute la commande **backup**. Malheureusement, le script ne fonctionne pas à cause du fait qu'à chaque fois qu'on extrait le fichier depuis l'archive du serveur, le **mtime** original est modifié par le temps d'extraction ce qui signifie qu'on ne va jamais pouvoir comparer les **mtime**. On a essayé plusieurs méthodes de compression pour conserver le **mtime** original ou même accéder les métadonnées du fichier, mais sans succès.

Voici la partie importante du script :

```
21              if tar -tf "$dest_dir/$archive_name" "./$filename" &> /dev/null; then
22                      # dest file exists
23                      source_mtime=$(stat -c %Y "$file")
24                      dest_mtime=$(tar -xf "$dest_dir/$archive_name" -O "./$filename" | stat -c %Y -)
25
26                      if [ "$source_mtime" -gt "$dest_mtime" ]; then
27                              # source file is newer, copy it
28                              cp "$file" "$temp_dir/$filename"
29                              echo "update : $filename"
30                      fi
31              else
32                      # dest file doesn't exist, copy it
33                      cp "$file" "$temp_dir/$filename"
34                      echo "copied : $filename"
35              fi
```

Avec la **ligne 21,** on regarde si un fichier spécifique existe. Ce n'est pas une extraction du fichier, juste une vérification.
Si le fichier existe, on prend le **mtime** du même fichier du côté client et du côté serveur et on récupère leur **mtime**.

→ **Pour le fichier côté client**
source_mtime=$(stat -c %Y "$file")
'stat' → affiche plusieurs informations sur le fichier
'-c' → spécifier le format de sortie de stat
'%Y' → obtenir l'heure de modification (**mtime**) du fichier en secondes

→ **Pour le fichier côté serveur**
dest_mtime=$(tar -xf "$dest_dir/$archive_name" -O "./$filename" | stat -c %Y -)
Ce que nous faisons ici, c'est extraire le fichier `$filename` de l'archive située dans le dossier de destination avec la commande `tar -xf`. Ensuite, nous redirigeons la sortie standard (qui est le fichier) avec `-O` vers le **pipe** qui le prend et l'utilise pour la commande suivante. Le `-` à la fin signifie que l'entrée est lue via la sortie standard envoyée par le **pipe**. En redirigeant la sortie avec `-O`, le fichier n'est pas réellement enregistré sur le disque.

→ **README.txt**
Un petit fichier texte avec quelques informations pour l'utilisateur comme : installation, description et utilisation des commandes, etc.

→ **logs.txt**
Un fichier logs qui est utilisé pour voir les actions faites par l'utilisateur. Ce fichier devrait être sur le serveur mais chaque fois qu'on exécute un script on doit saisir 4-5 le mot de passe pour envoyer chaque ligne de log dans le logs.txt du serveur donc pour le moment il est sur la machine du client.

Voici un extrait du fichier logs.txt

```
[2023/06/18 15:45:01]: (backup.sh) Executing backup.sh script
[2023/06/18 15:45:01]: (backup.sh) Backup of directory : /sourceFolder
[2023/06/18 15:45:01]: (backup.sh) BACKUP COMPLETED
[2023/06/18 15:45:40]: (backup.sh) Executing backup.sh script
[2023/06/18 15:45:40]: (backup.sh) Backup of directory : /sourceFolder/
[2023/06/18 15:45:40]: (backup.sh) BACKUP COMPLETED
[2023/06/18 15:57:48]: (backup.sh) Executing backup.sh script
[2023/06/18 15:57:48]: (backup.sh) Backup of directory : /sourceFolder/
[2023/06/18 15:57:49]: (backup.sh) BACKUP COMPLETED
[2023/06/18 16:07:26]: (init.sh) Moving current directory to /backup
[2023/06/18 16:07:26]: (init.sh) Adding alias backup='/backup/backup.sh' in .bashrc
[2023/06/18 16:07:26]: (init.sh) Adding alias restore='/backup/restore.sh' in .bashrc
[2023/06/18 16:07:26]: (init.sh) Cron entry added to crontab
[2023/06/18 16:07:26]: (init.sh) init.sh script executed successfully
[2023/06/18 16:11:32]: (backup.sh) Executing restore.sh script
[2023/06/18 16:11:34]: (backup.sh) List of backups
[2023/06/18 16:12:07]: (backup.sh) Executing backup.sh script
[2023/06/18 16:12:07]: (backup.sh) Backup of directory : /sourceFolder
[2023/06/18 16:13:06]: (backup.sh) BACKUP COMPLETED
[2023/06/18 16:14:35]: (backup.sh) Executing backup.sh script
[2023/06/18 16:14:35]: (backup.sh) Backup of directory : /sourceFolder/
[2023/06/18 16:14:36]: (backup.sh) BACKUP COMPLETED
[2023/06/18 16:14:51]: (backup.sh) Executing backup.sh script
[2023/06/18 16:14:51]: (backup.sh) Directory doesn't exist
```

→ **init.sh**
Initialiser / installer le système de sauvegarde.

→ **backup.sh**
Sauvegarder un dossier et de l'envoyer au serveur.

→ **restore.sh**
Restauration des backups faites.

*** Pour plus d'informations sur le code voir **V. SCRIPTS.**

IV. COMMANDES

Notre système de sauvegarde consiste en deux commandes.

► `$ backup`
La commande `$ backup` a comme but de sauvegarder les fichiers contenu dans dans un répertoire. Il peut être utilisé de deux façons différentes.

1. `$ backup`
Fait une sauvegarde du répertoire dans lequel on travaille actuellement

2. `$ backup /backupFiles`
Fait une sauvegarde du répertoire choisi

► `$ restore`
La commande `$ restore` a comme but la restauration des sauvegardes. Il peut être utilisé de deux façons différentes.

1. `$ restore`
Nous affiche la liste des sauvegardes qu'on a fait / qu'on veut récupérer

2. `$ restore backup_2023-06-18_165213.tar.gz`
Récupérer la sauvegarde choisi

Je profite de cette situation pour vous aussi expliquer comment les sauvegardes sont nommées.
Toutes les sauvegardes commence par **backup_**
Ensuite, on a la date et l'heure auxquelles le script a été exécuté.
Dans cet exemple l'heure est **16 h : 52 min : 13 sec**

V. SCRIPTS

Dans cette section on va voir le contenu des scripts et quelques explications.
Les éléments qui se répètent dans plusieurs scripts ne vont pas être expliqués. Ceci s'appliquent aux fonctions ou commandes. Si vous voulez voir l'entièreté des scripts voici le lien du gitea :
https://ytrack.learn.ynov.com/git/mmihail/infra_sauvegarde/

► init.sh ◄

```
1  #!/bin/bash
2
3  #
4  # Variables
5  #
6
7  commands=("alias backup='/backup/backup.sh'" "alias restore='/backup/restore.sh'")
8
```

Toutes les scripts bash commencent par `#!/bin/bash`

Ligne 7 → Déclaration d'un variable `commands`. C'est un tableau contenant deux lignes de codes qui vont être inséré dans le fichier `.bashrc`

```
9   #
10  # Functions
11  #
12
13  # Add log entry to logs file
14  function log_entry() {
15          echo "$(date +"[%Y/%m/%d %H:%M:%S]"): (init.sh) $1" >> /backup/logs.txt
16  }
```

Ligne 14 → Déclaration d'une fonction en bash
Ligne 15 → `$ echo` … >> …
Ceci prend le résultat du `$ echo` et l'envoie dans un autre fichier (ici c'est logs.txt)
Ligne 15 → $(date +"...")
Toute ligne qui s'écrit de cette façon `$()` s'appelle une substitution de commande. Dans un sens, c'est comme une fonction.

%Y → l'année (year)
%m → le mois (month)
%d → le jour (day)

%H → l'heure (hour)
%M → minutes (minute)
%S → secondes (seconds)

Comment vous pouvez aussi voir, on peut mettre des **/** ou des **:** entre les **%**

```
18  # Move dir to /backup
19  function move_dir() {
20          mv $(pwd) /backup
21          log_entry "Moving current directory to /backup"
22  }
```

Ligne 20 → La commande `$ mv` déplace un répertoire vers un autre. On peut aussi changer le nom. Ici on prend le répertoire qu'on est dedans et on le met dans `/backup`
Ligne 21 → Pour appeler une fonction qu'on a fait il suffit de juste taper son nom. Pas besoin de parenthèses ou de saisir les paramètres dans ces parenthèses. Ici on

appelle la fonction qui envoie un message aux logs.txt

```
24  # Add alias to .bashrc
25  function add_alias() {
26          for command in "${commands[@]}"; do
27                  if grep -Fxq "$command" ~/.bashrc; then
28                          echo "Alias exists. Skipping..."
29                          log_entry "$command already exists in .bashrc"
30                  else
31                          echo "$command" >> ~/.bashrc
32                          echo "Alias copied to .bashrc"
33                          log_entry "Adding $command in .bashrc"
34                  fi
35          done
36  }
```

Ligne 26 → On réalise une boucle `for` où `command` est la variable qui va contenir tous les éléments du tableau `commands`. Pour accéder à ses éléments il faut faire `${nomDuTableau[@]}`. Les `for` est les `if` se terminent par un `;` pour signifier la fin. Ensuite on utilise un mot “spécial” qui peut changer en fonction de qu’est-ce qu’on utilise. Dans une boucle `for`, le mot est `do` et pour la fin c’est un `done`. Alors que pour un `if` le mot utilisé est `then` et se finit par `fi` (on peut aussi utiliser `else` si on a besoin)

Ligne 27 → On réalise une condition `if`. On utilise le mot clé `grep` avec les options `-Fxq`. Pour résumer. On fait une recherche de la variable `$command` (de la boucle `for`) et on cherche si ceci existe dans le fichier `.bashrc`.

‘grep’ → commande utilisé pour rechercher des patterns
‘-F’ → indique de traiter le recherche de `$command` comme un texte simple, caractère par caractère
‘-x’ → indique que la ligne entière doit correspondre exactement à la ligne spécifié par `$command`
‘-q’ → indique à `grep` de ne pas afficher les résultats. Dans ce cas on a juste besoin du résultat de la condition.

Ligne 28-29 → Si la condition est vraie, on n’a plus besoin d’insérer la ligne donc on indique que la ligne existe déjà.

Ligne 31-33 → Si la condition est fausse, on envoie la commande dans .bashrc et on indique au utilitaire que la ligne a été ajoutée.

```
38  # Add cron entry crontab
39  function add_crontab() {
40          if crontab -l | grep -q "/backup.sh"; then
41                  echo "Cron entry exists. Skipping..."
42                  log_entry "Cron entry already exists"
43          else
44                  (crontab -l ; echo "50 20 * * * /backup/backup.sh /sourceFolder") | crontab -
45                  echo "Cron entry added to crontab"
46                  log_entry "Cron entry added to crontab"
47          fi
48  }
```

Ligne 40 → C'est une condition `if`. On affiche le contenu du crontab avec `crontab -l` et en utilisant le **pipe** `|`, on envoie le résultat (donc le contenu du crontab) pour faire une recherche si **"/backup.sh"** existe.

Ligne 41-42 → Si la condition est vraie, on n'a plus besoin d'insérer la ligne donc on indique que la ligne existe déjà.

Ligne 44 → Dans ces parenthèses on exécute 2 commandes (séparer par le `;`)
La première affiche le contenu du crontab et la deuxième affiche un texte avec l'aide de echo. Ensuite, on envoie tout cela à l'aide du **pipe** dans le crontab qui va tout remplacer par le résultat du pipe. Le tiret `-` signifie que l'entrée provient du **pipe**.

50 20 * * * /backup/backup.sh /sourceFolder

'50' → indique les minutes
'20' → indique l'heure
'* * *' → indique le jour / mois / année. Ici le * indique le fait que le script va être exécuté chaque jours de chaque mois de chaque année (donc tout le temps)

'backup/backup.sh /sourceFolder' → exécution du script **backup.sh** mais en ajoutant le dossier qu'on veut sauvegarder. Comme ce script va être exécuté en arrière-plan, on ne pourra pas vraiment utiliser le `$(pwd)` donc on doit saisir un répertoire.

Ligne 44-46 → Si la condition est fausse, on insère la ligne dans le crontab.

► backup.sh ◄

```
3   #
4   # Variables
5   #
6
7   send_backup=0
8   current_dir=$(pwd)
9   chosen_dir=$1
10  archive_name="backup_$(date +"%Y-%m-%d_%H%M%S").tar.gz"
11  temp_dir=$(mktemp -d)
```

Ligne 9 → le `$1` indique le paramètre envoyé. Si on fait `backup /backupFiles`, alors le `$1` correspond au `/backupFiles`.
Ligne 11 → On créer un dossier temporaire pour mettre tous les fichiers qu'on veut sauvegarder pour ensuite tout envoyer sur le serveur.

```
13  #
14  # Remote Server Details
15  #
16
17  remote_ip="192.168.64.5"
18  remote_user="root"
19  remote_directory="/serverBackup"
```

Ligne 17 → on indique l'adresse ip du serveur pour pouvoir se connecter
Ligne 18 → on indique l'utilisateur qu'on veut se connecter avec
Ligne 19 → on indique le répertoire où on veut stocker les sauvegardes sur le serveur

*** Ici on utilise **root** mais pour limiter les risques c'est mieux d'utiliser un utilisateur (du côté client et du côté serveur) avec le moindre de permissions possibles pour limiter tout risque.

```
21  # SSH Agent
22  # private_key=~/.ssh/id_rsa
23
24  # eval $(ssh-agent)
25  # ssh-add $private_key
```

Ceci est une tentative d'essayer de résoudre le problème d'exécution automatique du script **backup.sh** avec crontab. On a essayé d'utiliser un agent pour utiliser la clé privée directement dans le code pour supprimer le besoin de saisir le mot du passe de la clé privée du ssh. L'agent fonctionne mais ça ne supprime pas le fait qu'on doit

saisir le mot de passe ce qui signifie l'authentification va échouer et donc la sauvegarde ne pas être envoyé alors que le script est bien exécuté (on peut le voir dans le fichier logs.txt)

```
36  # Backup directory
37  function backup_dir() {
38          dir=$1
39          file_count=$(find $dir -type f | wc -l)
40
41          if [ $file_count -ne 0 ]; then
42                  cp -R "$dir"/* "$temp_dir"
43                  send_backup=1
44                  log_entry "Backup of directory : $dir"
45          else
46                  echo "NO FILES DETECTED IN $dir"
47                  log_entry "No files detected in $dir"
48          fi
49  }
```

Ligne 39 → Chercher dans un répertoire les fichiers de type -f (donc file) puis les compter.

$(find $dir -type f | wc -l)

'find' → faire une recherche
'$dir' → le répertoire où on veut faire une recherche
'type -f' → fichier de type -f (file)
'**Wc**' → compter (word count)
'-l' → indique qu'on va compter les lignes

Ligne 41 → Si le nombre de lignes n'est pas égal (`-ne` not equal) à 0 on fait une sauvegarde.
Ligne 42 → on fait une copie avec `cp` avec l'option `-R` qui va copier les fichiers ainsi que les dossiers.

L'astérisque (*) représente un joker (wildcard). C'est ce qui nous permet de parcourir chaque fichier dans le répertoire. Sans lui, (/), le répertoire serait traité comme un seul fichier. De plus, en utilisant /**, on parcourt également les fichiers du dossier ainsi que les sous-répertoires

```
51  # Send backup to server
52  function send_backup_to_server() {
53          send=$1
54
55          if [ "$send" -eq 1 ]; then
56                  tar -czf "$temp_dir/$archive_name" -C "$temp_dir" .
57                  scp "$temp_dir/$archive_name" ${remote_user}@${remote_ip}:${remote_directory}/
58                  # for the ssh agent add scp -i $private key ...
59
60                  rm -rf "$temp_dir"
61                  echo "Backup completed"
62                  log_entry "BACKUP COMPLETED"
63          fi
64  }
```

Ligne 55 → `-eq` signifie égal à (equal)
Ligne 56 → cette ligne de code crée une archive compressée (.tar.gz) à partir des fichiers et répertoires présents dans le répertoire `$temp_dir`. L'archive est enregistrée avec le nom spécifié dans `$archive_name` dans le répertoire `$temp_dir`.

tar -czf "$temp_dir/$archive_name" -C "$temp_dir" .

'tar' → manipuler les archives
'-c' → créer une archive (create)
'-z' → compression avec gzip pour réduire la taille
'-f' → spécifie le nom du fichier
'-C "$temp_dir"' → indique le répertoire duquel les fichiers / répertoires seront inclus dans l'archive
'.' → indique le répertoire courant

```
72  if [ -z "$chosen_dir" ]; then
73          backup_dir $current_dir
74  else
75          if [ -d "$chosen_dir" ]; then
76                  backup_dir $chosen_dir
77          else
78                  echo "THE DIRECTORY DOESN'T EXIST"
79                  log_entry "Directory doesn't exist"
80          fi
81  fi
```

Ligne 72 → `-z` vérifie si la variable est vide
Ligne 75 → `-d` vérifie si le répertoire (directory) existe

► restore.sh ◄

```
27  # Show the list of backups
28  function find_backups() {
29    ssh ${remote_user}@${remote_ip} "bash -s" << EOF
30      file_count=\$(find ${remote_directory} -name '*.tar.gz' | wc -l)
31
32      if [ \$file_count -ne 0 ]; then
33        number=1
34        for archive in "${remote_directory}"/*; do
35          if [ -f "\$archive" ]; then
36            archive_name=\$(basename "\$archive")
37            echo "\$number. \$archive_name"
38            ((number=number+1))
39          fi
40        done
41        printf "\nTYPE THE ARCHIVE NAME THAT YOU WANT TO RESTORE\n"
42      else
43        echo "YOU MUST CREATE A BACKUP BEFORE SEEING THE LIST OF BACKUPS"
44      fi
45  EOF
46  }
```

Ligne 28 → On se connecte avec ssh (avec l'aide de notre user et adresse IP qu'on a déclarer avant) au serveur.

Ligne 30-44 → Une fois connecté on va exécuter tout ce code là dans le terminal du serveur.

ssh ${remote_user}@${remote_ip} "bash -s" << EOF

'"bash -s"' → lance un bash shell à l'intérieur du serveur pour pouvoir exécuter des commandes.
'<< EOF' → cela s'appelle une **"here document".** Il nous permet de fournir des entrées multiples. L'indicateur **EOF** est utilisé pour marquer le début et la fin du bloc d'entrée.

```
48  # Get the chosen backup
49  function get_backup() {
50    if ssh ${remote_user}@${remote_ip} "test -f '${remote_directory}/${name_archive}'"; then
51      scp ${remote_user}@${remote_ip}:"${remote_directory}/${name_archive}" . > /dev/null
52      echo "Backup was copied to current directory: $(pwd)"
53    else
54      echo "THE BACKUP DOESN'T EXIST"
55    fi
56  }
```

Ligne 50 → if ssh ${remote_user}@${remote_ip} "test -f '${remote_directory}/${name_archive}'"; then

‘"test -f '${remote_directory}/${name_archive}'"’ → c’est la commande à exécuter sur le serveur si la sauvegarde qu’on essaye de récupérer existe. C’est le `test -f` qui vérifie cela

Ligne 51 → scp ${remote_user}@${remote_ip}:"${remote_directory}/${name_archive}" . > /dev/null

‘scp’ → C'est la commande utilisée pour copier des fichiers entre notre client et le serveur
‘:"${remote_directory}/${name_archive}"’ → C’est l’endroit du fichier qu’on essaye de récupérer
‘**. > /dev/null**’ → C’est une redirection de sortie qui redirige la sortie standard (stdout) vers le fichier `/dev/null`. Ceci signifie que le résultat de la commande ne va pas être affiché.