# BUT DU PROJET

Réaliser un serveur et un client `ssh`, de nom `myssh`, capables d'éxécuter des programmes sur une machine distante.

Votre projet sera composé des éxécutables suivants :

— `mysh` : le shell de votre projet ;
— `myssh` : le client SSH ;
— `mysshd` : le daemon SSH permettant d'accepter des connexions SSH ;
— `myssh-server` : un serveur ssh exécutant votre `mysh`, lisant les commandes depuis une socket et renvoyant les résultats dans la socket ;
— `myls` : une commande **externe** à votre shell, équivalente à `ls -l` ;
— `myps` : une commande **externe** à votre shell, équivalente à `ps aux` ;

L'ensemble des fonctionnalités de ce `ssh` est décrit dans les paragraphes suivants.

## MYSH

## 1. Lancement de commandes

La première fonctionnalité est de pouvoir exécuter des programmes lus depuis une invite de commandes indiquant le répertoire courant (~ représentera la *home directory* de l'utilisateur). Votre `shell` attendra la fin de l'exécution de la commande avant de réafficher son invite de commandes.

### a. Séquencement

Les commandes pourront être enchainées inconditionnelement par des `;`.

```
> ls; cat /etc/passwd
```

Elles pourront également être enchaînées conditionnellement avec les opérateurs `&&` et ||. La commande suivant un `&&` (respectivement ||) est éxécutée si et seulement si la commande le précédant a réussi (respectivement échouée).

```
> gcc -o mysh myshell.c && .mysh
> test -d .can || mkdir .can
```

### b. Wildcards

Les lignes de commandes de votre shell devront remplacer les caractères “jokers” par leur(s) correspondances(s) calculée(s) à partir du répertoire courant. Les caractères à gérer sont :

— `*` : qui se remplace par une suite quelconque de caractère. Cette suite de caractères peut éventuellement être vide ;
— `?` : qui se remplace par exactement un caractère ;
—

```
> cat *.[ch]
> ls ../[A-Z.]*[^~]
> wc -l /etc/?????
```

Dans l'exemple précédent, la première ligne de commandes permet de visualiser le contenu de tous les fichiers se terminant par l'extension `.c` ou `.h`. La deuxième ligne permet de lister toutes les entrées du répertoire père qui commence par une majuscule ou par un point et qui ne se terminent pas par ~. La dernière ligne permet de compter les lignes de tous les fichiers dont le nom comporte exactement 5 caractères et qui se situent dans /etc. Il est à noter que les caractères “jokers” précédés d'un \ ne sont pas remplacer.

## 2. Commandes

### a. Changer de répertoire

Afin de pouvoir se balader dans l'arborescence de fichiers, votre shell, devra disposer d'une commande interne `cd` permettant de se déplacer dans le répertoire passé en paramètre de cette commande. Si aucun répertoire n'est donné, le déplacement se fera vers la `home directory` de l'utilisateur courant du shell.

### b. Sortie du shell (et propagation du CTRL+c)

Deux moyens seront offerts pour quitter votre shell. La commande `exit` qui quittera votre shell sans tuer les commandes lancées en tâche de fond (background, voir section 4 ) et le `CTRL+C` qui demandera une confirmation avant de quitter mais qui tuera tous les processus en tâches de fond.

Il est à noter que si le `CTRL+C` survient alors qu'il y a une commande en cours d'éxécution (en foreground), ce `CTRL+C` devra alors être propagé au processus en cours d'éxécution et ne sera pas considéré comme une demande de sortie de votre shell.

### c. Code de retour d'un processus

Votre shell intégrera la commande interne `status` qui affiche pour le dernier processus (`xxxx`) exécuté en premier plan (en foreground) :

— `xxxx terminé avec comme code de retour YYY` si le processus s'est terminé normalement ;
— `xxxx terminé anormalement` lors d'une terminaison anormale (comme par exemple l'interruption via un CTRL+c).

### d. Lister le contenu d'un répertoire `myls`

En programme **externe** à votre shell, il vous est demandé également de programmer une commande de nom `myls` équivalente à la commance `ls -l` et qui acceptera deux options `-a` et `-R`. Ces options permettent respectivement de lister les fichiers cachés et d'explorer récursivement le ou les répertoires passés en paramètre de la commande. Votre `myls` sera par défaut en couleur, une couleur sera attribuée à chaque type de fichier. Les options peuvent être combinées. Si aucune entrée à lister n'est passée en paramètre, par défaut le répertoire courant est listé. Voici quelques exemples de commandes valides qui devront être gérées :

```
> myls / .. foo bar duc
> myls -aR /
> myls -R
> myls -R -a /
> myls -Ra ~
```

### e. Afficher l'état des processus en cours

Egalement, en programme **externe**, vous devez réaliser la commande `myps` équivalente à la commande `ps aux` à la différence que votre programme affichera en couleur les processus en fonction de leur état.

## 3. Les redirections

### a. Les tubes

Vos lignes de commandes devront permettre le "pipeline" de commandes au travers du symbole |. Ce symbole permet de rediriger la sortie standard du programme précédant le | vers l'entrée standard du programme suivant |. Les commandes pourront être "pipelinées" à l'infini.

```
> ls | sort -r
> ps | grep mysh | wc -l
```

### b. Redirections vers ou depuis un fichier

Vous devrez gérer les redirections usuelles vers ou depuis les fichiers à savoir :
— > redirige la sortie standard de la commande vers le fichier. Le précédent contenu du fichier est perdu ;
— >> redirige la sortie standard de la commande vers le fichier. L'écriture s'effectue en fin de fichier, le précédent contenu du fichier est donc conservé ;
— 2> idem que > pour la sortie standard des erreurs ;
— 2>> idem que >> pour la sortie standard des erreurs ;
— >& redirige la sortie standard et la sortie standard des erreurs de la commande vers le fichier. Le précédent contenu du fichier est perdu ;
— >>& redirige la sortie standard et la sortie standard des erreurs de la commande vers le fichier. Le précédent contenu du fichier est conservé ;
— < redirige l'entrée standard de la commande vers `stdin`.

Il est à noter que, quelle que soit la redirection utilisée, la syntaxe est toujours de la forme : `commande REDIRECTION fichier`

```
> find . type f -name \*.mp3 >> /home/ens/mazure/listofsongs
> nl < myshell.c
```

La premire commande ajoute au fichier `listofsongs` de `/home/ens/mazure` les fichiers dont l'extension est `.mp3` du répertoire courant. La seconde commande affiche les lignes numérotées du fichier `myshell.c`. La troisième commande récupère tous les fichiers d'inclusion (exceptés ceux des librairies standards) du fichier `myshell.c`.

## 4. Premier et arrière plans

Jusqu'à présent toutes les commandes exécutées, l'étaient en premier plan ou en "foreground", c'est-à-dire que l'on attendait la fin de l'exécution de l'ensemble de la ligne de commandes avant d'afficher de nouveau l'invite de commandes. Votre shell devra prévoir le lancement de commandes en arrière-plan ou en "background", c'est-à-dire que l'invite de commandes sera affichée sans attendre la fin de la commande. Pour cela un `&` suivra la commande à lancer en background.

```
> emacs &
> ls -lR | gzip > ls-lR.gz &
```

Au lancement d'une commande en arrière plan, avant de réafficher l'invite de commandes, votre shell affichera une ligne de la forme : `[xxx] yyy` où `xxx` représente le numéro du "job" en "background" et `yyy` le pid du processus. Le numéro de job est un compteur qui est réinitialisé lorsque plus aucun job n'est exécuté en arrière plan.

Lorsqu'une commande lancée en arrière plan se termine, on affichera : `zzz (jobs=[xxx], pid=yyy) terminée avec status=sss`

où `zzz` la commande lancée, `xxx` le numéro du job, `yyy` le pid et `sss` le code de retour de la commande (la valeur -1 sera affichée si la commande s'est terminée anormalement).

### a. Commande `myjobs`

La commande interne `myjobs` permettra d'afficher la liste des processus en arrière plan. L'affichage se fera de la manière suivante : `[xxx] yyy Etat zzz` où `xxx` représente le numéro de job, `yyy` le pid, `Etat` l'état du processus qui pourra être *En cours d'exécution* ou *Stoppé* et `zzz` la commande lancée. Un job par ligne sera affiché.

### b. Passer une commande de foreground à background et inversement

Lorsqu'une commande est lancée en foreground, il est possible de l'interrompre en lui propageant le signal envoyé par `CTRL+Z` (signal `SIGSTP`). La commande est alors stoppée, votre shell reprend la main en indiquant que la commande `zzz` devient le job `xxx` et qu'il est stoppé.

Les commandes internes `myfg` et `mybg` permettent de modifier l'état d'un job. Ces commandes peuvent admettre un numéro de job en paramètre. La commande `mybg` permettra de passer le job stoppé ou en foreground de plus grand numéro ou de numéro passé en paramètre en exécution en arrière plan.

Inversement, la commande `myfg` permettra de passer le job stoppé ou en background de plus grand numéro ou de numéro passé en paramètre en exécution en premier plan. Le signal `SIGCONT` permet la demande de reprise d'un processus précédemment stoppé. Un job en background n'a pas besoin d'être stoppé pour repassé en foreground. Si la commande `mybg` s'applique à un numéro de processus déjà en background, une erreur est signalée et aucun changement n'est réalisé sur les jobs.

## 5. Les variables

Pour être un interpréteur de commande qui se respecte votre shell doit intégrer la prise en charge de variables. Deux types de variables sont à gérer : les variables locales et d'environnement. Pour les variables d'environnement voir la sous-section dédiée dans la section `myssh-server`.

Les variables locales sont locales à votre shell, c'est à dire que deux exemplaires de votre shell peuvent avoir un même nom de variable avec une valeur différente dans chaque shell.

Pour créer une variable locale, la commande sera `set`, pour la détruire la commande sera `unset`.

```
> set a=foo
> echo $a
foo
> mysh
> echo $a

> a=bar
> echo $a
bar
> exit
> echo $a
foo
```

Vous aurez également des variables d'environnement à gérer voir la section *MYSSH-SERVER*.

# MYSSH

`myssh` est votre client `ssh`. Il devra accepter un nom d'utilisateur et un host de la forme : `nom_utilisateur@host`.

## 1. Identification

La première étape est de gérer l'identification de l'utilisateur sur la machine distante. Si tout se passe bien il affichera un `prompt` afin de lire les commandes utilisateurs. Elles ne devront pas être interprétées localement mais envoyées à `host` pour qu'il interprète et éxécute les commandes.

```
~> myssh toto@10.10.10.1
password:
toto@host1: ~>
```

Si échec d'authentification :

```
~> myssh toto@10.10.10.1
password:
Invalid User/password for host1
```

## 2. Configuration

L'utilisateur pourra créer un fichier `config` dans le dossier `.myssh` de son home directory et ainsi préciser des raccourcis de connexion.

```
Host host1
    Hostname 10.10.10.1
    User toto
```

L'exemple précédent précise un alias vers l'adresse ip `10.10.10.1` et précise un nom d'utilisateur. Ainsi il est possible d'écrire maintenant :

```
~> myssh host1
password:
toto@host1: ~>
```

### 3. Sortie du shell (et propagation du CTRL+c)

Si votre `myssh` reçoit un `CTRL+C` il transmettra le message au serveur qui proposera le même comportement que dans la section 1.2.b (voir la section dans le protocole sur les signaux pour la manière d'envoyer l'information de la réception d'un signal).

## MYSSHD

Ce programme est un `daemon` acceptant une nouvelle connexion. Avant de lancer un shell pour l'utilisateur distant, il se chargera de vérifier que l'utilisateur peut s'authentifier correctement. Si tout se passe bien il lancera un `myssh-server` qui pourra "vivre sa vie".

## MYSSH-SERVER

Ce programme est un shell `mysh` capable de lire des commandes depuis une socket et de renvoyer le résultat sur une socket. Il devra également gérer des variables d'environnement.

Les variables d'environnement sont stockées dans une zone de mémoire partagée entre les machines où l'utilisateur se connecte. Cette zone sera initialisée avec toutes les variables définies dans le tableau `envp` : `int main(int args, char *argv[], char* envp[])`. Le dernier exemple de votre `myssh-server` qui tourne à un instant donné doit détruire cet espace partagé avant de mourir.

```
~> myssh toto@host1
password:
toto@host1:~> setenv a=foo;

~> myssh toto@host2
password:
toto@host1:~> echo $a
foo
```

La variable créée sur la machine `host1` est disponible sur la machine `host2`.

```
toto@host1:~> exit

toto@host2:~> echo $a
foo
toto@host2:~> exit
```

Si l'on quitte la connexion vers `host1`, la variable est toujours disponible dans `host2`.

```
~> myssh toto@host1
password:
toto@host3:~> echo $a

toto@host3:~> exit
```

Après avoir quitté la dernière instance de votre ssh(`host2`), si l'on relance un nouveau ssh vers `host1` la variable n'est plus définie.

# PROTOCOLE DE COMMUNICATION SSH

Le protocole proposé dans ce projet est basé sur la spécification ssh d'openssh.

## 1. Généralité et format du protocole

Les messages envoyés seront composés de plusieurs champs pouvant être de type :
— `string` : une chaîne de caractères de taille arbitraire ;
— `boolean` : un octet pouvant prendre 0 qui équivaut à Faux ou 1 qui équivaut à Vrai ;
— `byte` : un entier représenté sur un octet, indiquant le type de message ;
Dans la suite lorsque vous verrez dans le format du message le champs `service name` la valeur sera `ssh`.

## 2. Authentification

L'authentification doit se faire par un échange de message entre le client et le serveur. Le client crée une requête d'authentification à laquelle le serveur répond.
Les types de messages possible pour l'authentification sont :
— `SSH_MSG_USERAUTH_REQUEST` qui vaut 50
— `SSH_MSG_USERAUTH_FAILURE` qui vaut 51
— `SSH_MSG_USERAUTH_SUCCESS` qui vaut 52

### a. Requête d'authentification

La requête d'authentification aura le format suivant :

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    method name
string    specific method fields
```

— `method name` peut dans la spécification openssh avoir les valeurs suivantes :
— "publickey"
— "password"
— "hostbased"
— "none"
Plus spécifiquement pour une requête d'authentification par mot de passe (`method name=passord`) le format sera :

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "password"
string    plaintext password
```

Dans notre projet nous ne gérerons que le protocole par `password`. Vous êtes libre de gérer le protocole `publickey` en vous basant sur les spécifications `openssh` se sera pris en considération en **bonus** dans votre projet. Il faudra alors le dire dans votre README et indiquer comment l'utiliser.

### b. Réponse du serveur

**Cas de l'échec d'authentification** Le serveur répondra le message suivant si l'authentification échoue :

```
byte SSH_MSG_USERAUTH_FAILURE
string message
```

Ici le message décrira la raison de l'échec et sera affiché par le client.

**Cas de l'authentification qui a réussi** Le serveur répondra le message suivant si l'authentification réussit :

```
byte SSH_MSG_USERAUTH_SUCCESS
```

## 3. Envoi d'une commande

Votre programme proposera deux modes d'envoi de commandes. Si l'utilisateur a précisé l'option `-c` alors la commande a exécuté sera celle précisée sur la ligne de commande et affichera le résultat. Dans le cas contraire votre programme ouvrira un prompt permettant d'exécuter la commande.

Les types de messages possibles pour les commandes sont :

— `SSH_MSG_CHANNEL_REQUEST` 98
— `SSH_MSG_CHANNEL_SUCCESS` 99
— `SSH_MSG_CHANNEL_FAILURE` 100

### a. Cas numéro 1 :

```
~> myssh toto@host1 -c ls
fichier1 fichier2
```

```
byte      SSH_MSG_CHANNEL_REQUEST
string    "exec"
string    command
```

où `command` n'est autre que la commande précisée par l'option `-c`.

La sortie standard de la commande a exécuter sera redirigée vers la socket. Lorsque la commande sera terminée le serveur répondra le message suivant en cas d'échec :

```
byte      SSH_MSG_CHANNEL_FAILURE
byte      code de retour du processus
```

Dans le cas d'un succès :

```
byte      SSH_MSG_CHANNEL_SUCCESS
byte      code de retour du processus
```

Le client affichera le message suivant : `processus distant terminé avec le code [xxx]` où `xxx` est le code de retour du processus.

### b. Cas numéro 2 :

```
~> myssh toto@host1
toto@host1:~>
```

```
byte      SSH_MSG_CHANNEL_REQUEST
string    "shell"
```

Ici la requête demande l'ouverture d'un `shell`. Le client proposera un prompt à l'utilisateur afin qu'il puisse rentrer ses commandes. Le prompt sera de la forme : `utilisateur@host:/path/distant>`.

Le serveur répondra :

```
byte      SSH_MSG_CHANNEL_SUCCESS
string    "shell"
```

La commande sera ensuite envoyée au serveur avec le message suivant :

```
byte      SSH_MSG_CHANNEL_REQUEST
string    "command"
```

où `command` correspond à la commande tapée par l'utilisateur dans le prompt.

La sortie standard de la commande a exécuter sera redirigée vers la socket. Lorsque la commande sera terminée le serveur répondra le message suivant en cas d'échec :

```
byte      SSH_MSG_CHANNEL_FAILURE
byte      code de retour du processus
```

Dans le cas d'un succès :

```
byte      SSH_MSG_CHANNEL_SUCCESS
byte      code de retour du processus
```

Le client affichera le message suivant : `processus distant terminé avec le code [xxx]` où `xxx` est le code de retour du processus.

### 4. Envoi d'un SIGNAL

Lorsque votre `myssh` recevra un signal il le transmettra à la machine distante. Le serveur doit soit prendre le signal pour lui si aucun programme ne tourne en premier plan soit transmettre le signal au programme tournant en premier plan.

Pour cela il enverra le message suivant :

```
byte      SSH_MSG_CHANNEL_REQUEST
string    "signal"
string    signal name (without the "SIG" prefix)
```

où `signal name` pourra avoir une des valeurs suivantes :

— ABRT
— ALRM
— FPE
— HUP
— ILL
— INT
— KILL
— PIPE
— QUIT
— SEGV
— TERM
— USR1
— USR2

## MODALITÉ DE REMISE DE PROJET

Le projet est à rendre par l'intermédiaire de Moodle au plus tard le 31 décembre minuit de l'année en cours. L'archive sera nommée : `[PROJET MYSSH] Nom_des_binômes`.

L'archive contiendra un README :

— en pourcentage la part de travail de chacun des binômes ;
— les fonctionnalités implémentées et celles non implémentées (faute avouée à moitié pardonnée) ;
— les "bugs" éventuels (idem), pire encore il est préférable de savoir que son programme "bugue" et de décrire les bugs en expliquant pourquoi vous n'avez pas réussi à les corriger que de ne pas savoir que votre programme bugue ce qui implique que vos tests unitaires n'étaient pas suffisants ;
— une petite blague pour détendre l'atmosphère (c'est optionnel) ;

## NOTE

Ce projet donnera lieu à deux notes : une en réseau et une autre en système.