Linux: Présentation du serveur X11

Table of Contents

- Linux: Présentation du serveur X11
- script xinitrc
  - Session terminal
  - Session complète
  - Sécurité
- Lancer des applications graphiques à distance
  - L'option "native"
    - vérification du serveur X11 en écoute
    - Utilisation de la transparence native de X11
    - Autorisation de tous les accès à la machine secondaire
  - L'option SSH
- Exécution d'un bureau distant complet
  - Utilisation de xinit pour l'image
  - Utilisation de pulsaudio pour le son

X11 est une technologie client/serveur. C'est à dire que lorsqu'une application est lancée, elle n'écrit jamais directement sur le périphérique graphique mais se connecte, par l'intermédiaire d'un kit graphique (Qt, GTK, etc) à un serveur d'affichage, le fameux Xorg exploitant le protocole X11.

Ainsi, à chaque fois qu'une application X11 veut afficher un rectangle, ou une image, c'est à ce serveur que cette demande est transmise et c'est lui qui effectue le rendu visuel. Dans l'autre sens, le serveur recueille les entrées telles la souris ou le clavier pour retransmettre cela à l'application.

Alors évidemment lorsque le client et le serveur sont sur une même machine, tout ceci ne va pas sans une certaine perte de performance. Cependant les impacts sur la vitesse sont limités à plusieurs étages comme par l'utilisation de sockets rapides (sockets UNIX) au lieu des classiques sockets sur IP, ou encore par un accès direct au GPU dans des cas critiques comme la 3D ou la vidéo.

Maintenant cette toute relative perte de performance ne doit pas faire oublier les immenses avantages de cette solution : la transparence réseau. En effet déporter l'affichage d'une application sur une machine distante, exécuter sur son écran local une application qui se trouve sur un serveur distant, tout cela et bien d'autres choses encore sont rendues possibles grâce au X11.

Pour la petite histoire il faut bien comprendre que cette transparence réseau n'est pas un coquetterie d'Unixien. En effet, à son origine, une architecture UNIX classique se composait d'un ou plusieurs serveur “puissant” et d'une myriade de petits terminaux graphiques causant couramment le X11 “en dur”, le tout sur un LAN 1Mb/s.

script xinitrc

~/.xinitrc est un script lu par la commande startx une fois que l'environnement graphique est chargé. Sa particularité est qu'il ne se termine pas immédiatement. Il doit rester actif tout au long de la session. S'il s'arrête, c'est la session graphique qui va elle aussi être stoppée pour retomber en console.

Session terminal

Une version simpliste de ce script consiste par exemple à lancer un terminal initial, généralement xterm. Ainsi, au lancement de la session, on disposera d'une console permettant de lancer d'autres applications. Pour fermer la session il suffit donc de fermer la console initiale.

```
#! /bin/bash

# console initiale
xterm
```

Pour tester cela, il faut arrêter lee gestionnaire de session (/etc/xinit.rc/gdm stop), s'authentifier dans une console, et enfin lancer la session graphique par la commande startx. Quasi instantanément vous doit voir apparaître xterm mais sans décoration de fenêtres.

En effet, par défaut, X ne décore car c'est le rôle d'un gestionnaire de fenêtres comme metacity. Si ce dernier est installé sur votre machine, on peut donc modifier votre script pour régler ce problème.

```
#! /bin/bash

# lancement du gestionnaire de fenêtres
metacity&

# console initiale
xterm
```

Metacity est lancé en arrière de plan, suivi par xterm. Maintenant au prochain startx, la console devrait être décorée.

Pour que tout fonctionne proprement il faut laner les autres composants graphiques tels que: DBUS et une session ConsoleKit dont le rôle est de fournir aux applications des informations utiles sur les différentes sessions. Cette brique est notamment utile à PolicyKit en charge de la gestion des autorisations par exemple sur les périphériques de stockages amovibles.

Pour lancer ConsoleKit et DBUS dans la même foulée, on va utiliser la commande ck-launch-session

```
#! /bin/bash
```

```
# Lancement groupé de dbus, metacity et xterm dans une session ConsoleKit
exec ck-launch-session dbus-launch metacity xterm & xtermpid=$!

# et on attends la mort de xterm
wait $wmpid
```

`ck-launch-session` va ainsi permettre de lancer une série de commandes. On récupère ici l'identifiant du processus de la dernière lancée (xterm) et on va attendre sa mort.

Session complète

A ce stade, on dispose d'un environnement complet. Il est bien évidement possible d'aller beaucoup plus loin, par exemple en lançant des applications par défaut, ou encore en opérant certains paramétrages

```
#! /bin/bash

# Lancement groupé de dbus, metacity et xterm dans une session ConsoleKit
exec ck-launch-session dbus-launch metacity xterm & xtermpid=$!

# chargement d'un fond d'écran (pour ceux qui aiment ça...)
feh --bg-center mon_beau_papier_peint&

# activation du pavé numérique
numlockx on&

# chargement d'un gestionnaire de presse papiers
parcellite&

# lancement de mutt
mutt &

# lancement d'IRSSI
irssi &

# et on attends la mort de xterm
wait $wmpid
```

Synchronisation sur le gestionnaire de fenêtres

On peut lancer un panel, une synchronisation avec unison, ou encore se débarrasser de xterm. Mais pour cela il faut un gestionnaire de fenêtre plus costaud que `Metacity` comme `WMFS` (il n'est pas dans les dépôts Debian mais se compile très facilement).

```
#! /bin/bash

# Lancement groupé de dbus, metacity et xterm dans une session ConsoleKit
```

```
exec ck-launch-session dbus-launch metacity wmfs & wmfspid=$!

# chargement d'un fond d'écran (pour ceux qui aiment ça...)
feh --bg-center mon_beau_papier_peint&

# activation du pavé numérique
numlockx on&

# Synchronisation de la sélection primaire et du presse papier
autocutsel -selection CLIPBOARD -fork
autocutsel -selection PRIMARY -fork

# lancement de mutt
mutt &

# lancement d'IRSSI
irssi &

# et on attends la mort de xterm
wait $wmfspid
~/.xinitrc minimaliste
```

Sécurité

Llancer X en mode console peut présenter un risque de piratage de la session. Un hacker peu en effet repasser en console via CTRL-ALT-F1 et arrêter X pour avoir ainsi accès au compte.

Pour y remédier, il suffit de verrouiller la console virtuelle après avoir lancé X. L'utilitaire `vlock` (`apt-get install vlock`) est conçu pour cet usage. Une fois installé, il suffit de lancer X de la manière suivante :

```
startx & vlock
```

De cette manière X se lance en tâche de fond et donne directement la main à `vlock` qui va verrouiller la console.

Pour compléter le dispositif, il est nécessaire de verrouiller aussi la session X locale, ce qui se fait cette fois par la commande `i3lock -d` (paquet `i3more`) ou encore `slock` (paquet `slock`).

Lancer des applications graphiques à distance

Aujourd'hui la transparence réseau offerte par X11 n'est même pas soupçonnée la majorité des utilisateurs de Linux qui ne travaillent que sur une et une seule machine. Mais X11 permet de couvrir un vaste ensemble de cas d'usage :

- ouvrir une session graphique complète sur une machine distante. Tout ce qui sera exécuté tournera sur la machine distante et s'affichera via le serveur X11 de la machine locale.
- ouvrir juste une application graphique, généralement via SSH, sur la machine distante.

Dans tous les cas le principe sera le même, une application est lancée à distance et se connecte à votre serveur X11 local

L'option "native"

Un serveur X11 utilise les sockets pour communiquer avec le client. Et pour que cette communication soit la plus rapide possible, il en utilise deux types : des sockets “lents” TCP/IP, et des sockets “rapides” UNIX. Dans le cas d'un serveur local parlant à une application locale, le socket utilisé est systématiquement de type UNIX.

Pour des raisons de sécurité, il y a de forte chance que votre serveur X soit paramétré de sorte à ne pas utiliser les sockets TCP/IP, et donc de ne pas permettre son utilisation par une machine distante. Nous allons donc changer cela en ayant conscience que cette manipulation n'est sure que sur un réseau local de confiance et protégé du réseau public.

Le lancement de votre serveur X11 est effectué par l'intermédiaire d'un gestionnaire de session, par exemple GDM pour Gnome. Pour autoriser les clients distant à utiliser ce serveur X11 via TCP, il faut donc lancer en tant que root l'outil gdmsetup, aller dans l'onglet sécurité, et décocher la ligne Refuser les connexions TCP au serveur X. Ceci fait vous pouvez fermer gdmsetup et relancer votre serveur X. Au retour, le port 6000 correspondant à l'écran X11 :0.0 (si c'est :1.0 le port sera 6001, et ainsi de suite) devrait être écouté par le serveur :

```
rootnetstat -anlp | grep X | grep -v STREAM
tcp        0      0 0.0.0.0:6000                0.0.0.0:*
LISTEN      5667/X
```

vérification du serveur X11 en écoute

Ceci fait, on va pouvoir se connecter sur la machine distante, par exemple avec SSH, et y lancer une commande graphique, par exemple evolution :

```
gaston@machine_distante>DISPLAY=mon_desktop:0.0 evolution
```

Utilisation de la transparence native de X11

C'est aussi simple que cela. Tout passe par l'utilisation de la variable `DISPLAY` qui indique à l'application où se trouve le serveur X11 à utiliser. En local, votre variable `DISPLAY` est sûrement `:0.0`. Le fait qu'il n'y ait rien devant le `:` indique au client que le serveur est local. Dans l'exemple il va donc aller sur la machine `mon_desktop`. `0.0` indique d'utiliser l'écran `.0` de la carte `0`.

Autorisation de tous les accès à la machine secondaire

Il est possible que le serveur soit configuré pour refuser les connexions externes et on aura alors l'erreur suivante :

```
gaston@machine_distante>DISPLAY=mon_desktop:0.0 evolution
No protocol specified
Impossible d'ouvrir l'affichage :
Exécuter « evolution --help » pour obtenir la liste complète des options en ligne de commande.
```

Ce qui compte ici c'est le `No protocol specified` qui dit, qu'on doit d'abord autoriser les connexions en lançant sur la machine secondaire la commande suivante :

```
gastonxhost +
access control disabled, clients can connect from any host
gastonssh machine_distante
gaston@machine_distante>DISPLAY=mon_desktop:0.0 evolution&
```

Et là, ça marche.

L'option SSH

SSH permet la redirection de ports, et plus particulièrement est capable de rediriger le socket UNIX d'un serveur distant sur un socket TCP/IP local. Cette caractéristique permet en toute simplicité de faire la même chose que précédemment, mais à travers une connexion chiffrée convenant mieux à un réseau LAN moins sécurisé et ce, sans aucun changement des paramétrage de sécurité.

Ainsi, utilisé avec l'option -X (certaines configuration font cela par défaut), ssh va ainsi automatiquement changer la variable `DISPLAY` pour pointer sur son “faux” serveur X11 local :

```
gaston@machine_distante>echo $DISPLAY
localhost:11.0
gaston@machine_distante>evolution&
```

Là, plus besoin d'autorisation via xhost ni d'écoutes de port TCP/IP, tout se fait automatiquement par SSH. Maintenant cette méthode n'est pas sans défaut. En effet, si la machine distante est un peu légère, le chiffrement temps réel de toutes les données circulant dans le tuyau SSH vont un peu pénaliser les performances. Mais sur un LAN moderne cela reste acceptable.

Concernant un réseau distant, par exemple à travers internet, c'est le volume de données échangé qui va pénaliser les performances. Il est donc plus sage dans ce cas d'opter sur une version compressée du protocole X11, comme nomachine/freeNX.

Exécution d'un bureau distant complet

Pour d'évidentes raisons de performances, du fait qu'on doive je travailler sur un LAN sécurisé, l'approche SSH n'apporte ici que peu d'avantage. Autant donc attaquer nativement le serveur X11 de sorte à bénéficier des meilleurs performances.

Utilisation de xinit pour l'image

Pour le gestionaire de fenêtre on utilise awesome. Le principe va donc être de créer un fichier `xinitrc-distant` qui va lancer à distance le gestionnaire de fenêtre.

```
xhost +
ssh ma-machine "DISPLAY=mon-portable:0.0 awesome"
```

- `xhost +` permet d'autoriser les accès distants au serveur X11 local. **Attention**: il est nécessaire pour que tout ceci fonctionne d'avoir activé le mode TCP/IP sur le serveur X11
- La ligne suivante permet de se connecter via SSH sur la machine distante pour lancer awesome. Le DISPLAY injecté en variable va quant à lui indiquer à awesome d'aller parler au serveur X11 de `mon-portable`.

il ne reste plus qu'à lancer la session dans la console linux :

```
xinit xinitrc-distant
```

orsqu'on tape **startx** dans la console Linux, c'est comme si on avait tapé `xinit ~/.xinitrc`. C'est commande **xinit** qui va concrètement lancer le serveur X11 et qui va ensuite exécuter le script **xinitrc**. Ici on indique donc juste un script alternatif à **xinit**.

Utilisation de pulsaudio pour le son

Comme X11, **PulseAudio** bénéficie d'une transparence réseau qui par défaut est limité à un socket UNIX. Comme pour X11 il faut donc commencer par activer le mode TCP/IP en modifianr le fichier `/etc/pulse/default.pa`. On doit y trouver une ligne normalement commentée `load-module module-native-protocol-tcp`. Décommenter la ligne, sauver le fichier et redémarrer la machine. Ceci fait, PulseAudio devrait être en écoute du port 4713 de la machine locale.

Il faut maintenant modifier notre fichier `xinitrc-distant` pour injecter dans l'environnement d'awesome la référence au serveur PulseAudio :

```
xhost +
```

```
ssh ma-machine "DISPLAY=mon-portable:0.0 PULSE_SERVER=mon-portable:4713
awesome"
```

Ceci fait, il ne reste plus qu'à relancer `xinit ~/xinit-distant` pour avoir maintenant accès à l'audio tant pour les entrées que les sorties.