Gérer sa session avec systemd

Table of Contents

De plus en plus de distributions passent à **systemd** comme alternative à l'**init de System V**.

L'un des avantages de **systemd** est d'avoir un système de services pour l'utilisateur, et c'est ce système qu'on va utiliser pour gérer une session.

Lorsqu'une session utilisateur démarre (que ce soit à distance via ssh ou en local), une instance de `systemd --user` démarre pour cet utilisateur. Cette instance a pour but de démarrer des services pour l'utilisateur, de façon similaire au processus 1 mais pour l'utilisateur. À noter que par défaut ces services se terminent quand la session de l'utilisateur disparaît.

Démarrage de systemd --user

Avant la version 206, c'était l'utilisateur qui pouvait choisir de démarrer (ou non) cette instance. Maintenant elle ne peut plus être démarrée par l'utilisateur et c'est le gestionnaire qui doit la lancer.

Elle est démarrée automatiquement à la connexion à condition que le module **pam_systemd** soit actif dans **pam** pour le type de session demandé. Pour cela, on ajoute une ligne de la forme

```
-session   optional   pam_systemd.so
```

(Le “-” indiquant que ce n'est pas essentiel à la session. Dit autrement : si ça plante c'est pas grave on continue quand même). Sous Archlinux, le fichier `/etc/pam.d/system-login` contient déjà cette ligne et concerne tous les types de connexion, locale ou distante.

Agencement de systemd

Systemd sépare les différents services en **slice**/**scope**/**service**, via l'utilisation de **cgroups** (une façon de regrouper un ensemble de processus et leurs éventuels descendants sans “échappatoire”

possible).

Un système typique ressemble à ceci (obtenu avec **systemd-cgls**) :

```
system.slice
    service_systeme1.service
    service_systeme2.service
user.slice
    user-1000.slice
        session-c1.scope
            Programme 1
            Programme 2
        session-c2.scope
        session-c3.scope
        user@1000.service
            systemd --user
            service_user1000_1.service
            service_user1000_2.service
            service_user1000_3.service
    user-1001.slice
        session-c4.scope
            Programme 1
            Programme 2
        session-c5.scope
        session-c6.scope
        user@1001.service
            systemd --user
            service_user1001_1.service
            service_user1001_2.service
            service_user1001_3.service
```

On remarque que les services de l'utilisateur sont démarrés dans un contexte distinct de la session. Il faudra donc faire attention à la propagation des variables d'environnement.

Dès que toutes les **session-c*.scope** d'un utilisateur sont quittées, le service **user@.service** correspondant s'arrête également, et avec tous les sous-services. C'est un point important à noter ! Par exemple, en l'état, utiliser les “timers” de **systemd** comme remplacement à **cron** ne marchera pas ! Heureusement, on peut passer outre et s'en sortir quand même.

Les services utilisateurs

Les services utilisateurs sont gérés exactement de la même façon que les services système, à ceci près que les unités (“unit”) sont recherchées dans des dossiers différents. Au lancement de `systemd --user`, c'est l'unité **default.target** qui est lancée et qui lancera à son tour les services nécessaires. systemctl –user et dbus

Les services gérés par **systemd** (qu'ils soient système ou utilisateur) se gèrent essentiellement via la commande `systemctl (--user)`, qui dépend fortement de **dbus**. La première chose à faire est donc de s'assurer que **dbus** tourne, aussi bien au niveau système (“system bus” pour dbus) qu'au

niveau utilisateur (“session bus”). Pour la partie système, on va se contenter d'ajouter une dépendance à **user@.service**. On va également normaliser le chemin du socket pour le “session bus” en l'indiquant dans la variable d'environnement appropriée **DBUS_SESSION_BUS_ADDRESS**. Comme **user@.service** est parent de tous les services de l'utilisateur, on va l'utiliser pour transmettre cette variable aussi :

```
# /etc/systemd/system/user@.service.d/dbus_env.conf
[Unit]
Wants=dbus.service

[Service]
Environment=DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/%I/bus
```

La ligne 2 du fichier indique qu'on souhaite que le service dbus système soit démarré, alors que la ligne 5 concerne la variable d'environnement pour l'utilisateur. Ce sont donc “deux bus” différents qu'on configure ici.

Dans ce qui suit, on va définir des différents services dont on aura besoin. Ils peuvent être définis ou activés globalement ou à la discrétion de l'utilisateur. Cela se fait soit selon l'endroit où ils sont placés (`/etc/systemd/user/` vs `$HOME/.config/systemd/user`) soit selon la façon dont ils sont activés (avec **–global** vs **–user**). Ici on définit les services et sockets associés à **dbus** globalement, mais ils seront ensuite activés ou non selon le choix de l'utilisateur :

```
# /etc/systemd/user/dbus.service
[Unit]
Description=D-Bus Message Bus
Requires=dbus.socket

[Service]
ExecStart=/usr/bin/dbus-daemon --session --address=systemd: --nofork --
nopidfile --systemd-activation
ExecReload=/usr/bin/dbus-send --print-reply --session --type=method_call --
dest=org.freedesktop.DBus / org.freedesktop.DBus.ReloadConfig
Restart=always
RestartSec=1
```

Lla principale différence avec le service système de même nom est dans l'utilisation de **–session** au lieu de **–system**

```
# /etc/systemd/user/dbus.socket
[Unit]
Description=D-Bus Message Bus Socket
Before=sockets.target
```

```
[Socket]
ListenStream=/run/user/%U/bus

[Install]
WantedBy=default.target
```

Pour rappel, **default.target** est la cible activée par défaut par systemd

À partir de là, l'essentiel est fait. On crée un fichier `mes_services.target` alias de `default.target`

```
# $HOME/.config/systemd/user/mes_services.target
[Unit]
Description=Mes services
Wants=dbus.service
AllowIsolate=true

[Install]
Alias=default.target
```

puis on crée des fichiers service ou autres cibles dans le même dossier. Ensuite, on les active avec la commande **systemctl**, fait habituellement pour gérer les services systèmes, sauf qu'ici on ajoute l'argument **–user**.

La directive **After** n'a pas le même sens pour un “service” et pour un “target”. Un service ne sera démarré qu'une fois que tous les services cités dans After sont prêts (pour la définition de “être prêt”, alors que pour un target il considère que les services cités font partie de lui (et peuvent donc être démarrés en même temps que d'autres services du target). Dans certains des cas présentés dans la suite cette distinction prend sens.

Des services utilisateur qui durent

Par défaut, les services associés à un utilisateur (tout ce qui est sous **user@1001.service** dans l'arborescence ci-dessus) sont arrêtés lorsque l'utilisateur se déconnecte de toutes ses sessions, notamment les **timers** qu'on aimerait utiliser comme remplacement à la **crontab**.

Passer outre se fait avec la commande suivante :

```
loginctl enable-linger user
```

Dans ce cas, **user@.service** sera démarré dès le boot pour l'utilisateur correspondant, même lorsque celui-ci n'est pas connecté.

L'astuce ci-dessus règle le problème d'avoir des crontab qui durent pour un utilisateur même lorsque celui-ci n'est pas connecté.

Pour les mails envoyés par les crontabs, les services de systemd loggent tout dans le journal, et aucun mail ne peut être programmé directement avec les services de systemd

Les variables d'environnement

Les variables d'environnement peuvent devenir un vrai casse-tête lorsqu'on utilise les services : comment transmettre les variables d'environnement nécessaires aux services, par exemple **GPG_AGENT_INFO** ou **SSH_AUTH_SOCK** ou même **DISPLAY**. Une des façons de faire est de spécifier dans le fichier du service une directive de la forme

```
Environment=DISPLAY=:0
```

ou

```
EnvironmentFile=/fichier/a/charger
```

Le problème est que cette configuration est statique et doit être définie pour chaque service ou globalement dans un fichier `/etc/systemd/system/user@.service.d/environment.conf`. On perd le “dynamisme” habituel des scripts de configuration.

On peut faire usage de la commande `systemctl --user import-environment` pour pallier ce problème. Cette commande permet d'importer des variables d'environnement, qui seront incluses dans les services démarrés par la suite. On commence par créer un service **setenv.service** qui va s'occuper de démarrer un script pour définir l'environnement:

```
# $HOME/.config/systemd/user/setenv@.service
[Unit]
Description=Set environment
Wants=dbus.service gpg-agent.service ssh-agent.service
After=dbus.service gpg-agent.service ssh-agent.service

[Service]
Type=oneshot
Environment=SSH_AUTH_SOCK=%t/ssh_auth_sock
ExecStart=%h/bin/systemd_setenv %i
```

et on active les services setenv@type.service selon les besoins. Ici par exemple, on a aussi défini une variable d'environnement pour le service. La raison est que la partie `"%t" (/run/user/1000/)` est plus facile à trouver dans ce fichier que dans un script… Il ne faudra juste pas oublier de l'appliquer dans le script ensuite.

Voici un exemple de script pour définir l'environnement :

```
# $HOME/bin/systemd_setenv
#!/bin/zsh

if [ "x$1" = "xcommun" ]; then
        . /etc/zsh/zprofile
        source $HOME/.gpg-agent-info
        export GPG_AGENT_INFO
        systemctl --user import-environment
        systemctl --user unset-environment PWD OLDPWD SHLVL _ MANAGERPID
elif [ "x$1" = "xgraphique" ]; then
        export XDG_CONFIG_HOME="$HOME/.config/"
        export SAL_USE_VCLPLUGIN=gtk
        export XDG_MENU_PREFIX="lxde-"
        export DISPLAY=:0
        systemctl --user import-environment XDG_CONFIG_HOME
SAL_USE_VCLPLUGIN XDG_MENU_PREFIX DISPLAY
fi
```

Ici on a deux environnements différents selon qu'on a démarré une session graphique ou juste une invite en ligne de commande. À noter que même si on quitte l'environnement graphique, les variables définies par ce moyen restent définies pour les services démarrés par la suite… On pourrait utiliser la commande

```
systemctl --user unset-environment VARIABLE1 VARIABLE2
```

de la même façon pour détruire les variables après utilisation (auquel cas il faut adapter le service **setenv** pour qu'il appelle un autre script — ou le même avec des arguments différents — lorsqu'on quitte l'interface graphique).

Les *-agents

Des services qui vallent le coup d'être démarrés, peu importe le type de session (graphique ou console ou distant) sont notamment les services **ssh-agent** et **gpg-agent**. Il faut cependant penser à transmettre les variables d'environnement aux processus intéressés.

Le service **envoy** peut être utilisé pour gérer ces agents, mais on ne peut pas séparer les deux agents avec.

Pour **ssh-agent**, on peut indiquer directement dans la commande d'exécution à quel endroit on souhaite mettre le socket, ce qui facilite les choses :

```
# $HOME/.config/systemd/user/ssh-agent.service
[Unit]
Description=ssh-agent
ConditionFileIsExecutable=/usr/bin/ssh-agent

[Service]
ExecStart=/usr/bin/ssh-agent -d -a %t/ssh_auth_sock
```

```
Restart=always

[Install]
WantedBy=mes_services.target
```

Pour **gpg-agent**, on ne peut que lui indiquer qu'on souhaite enregistrer les informations (les variables d'environnement) dans un fichier, et on ne peut pas forcer le socket à être placée à un endroit prévisible.

```
# $HOME/.config/systemd/user/gpg-agent.service
[Unit]
Description=gpg-agent
ConditionFileIsExecutable=/usr/bin/gpg-agent

[Service]
ExecStart=/usr/bin/gpg-agent --daemon --write-env-file %h/.gpg-agent-info
Type=forking
Restart=always

[Install]
WantedBy=mes_services.target
```

On peut ensuite charger le fichier `$HOME/.gpg-agent-info` pour avoir accès à l'agent.

```
source $HOME/.gpg-agent-info
```

Le fait de ne pas avoir un emplacement standard peut poser problème si par exemple l'agent redémarre : selon toute probabilité, le socket ne sera pas au même endroit par la suite et les programmes déjà lancés ne sauront pas où contacter l'agent.

Utiliser systemd pour gérer sa session graphique

Il peut être tentant d'utiliser systemd pour gérer également sa session graphique : définir son gestionnaire de fenêtre préféré comme un service et pouvoir l'arrêter et switcher vers un autre gestionnaire de fenêtre à volonté sans avoir à redémarrer sa session. On verra plus tard que cela peut poser de nouveaux problèmes.

Session graphique comme service

Le plus simple dans ce cas est de définir une unité **graphical.target** (ou **graphical@.target** si on veut définir plusieurs gestionnaires de fenêtre) et de l'activer dans le fichier `.xsession` (ou autre selon ce qui est exécuté pour la session).

Il peut être nécessaire d'avoir les variables d'environnement correctement définies à ce moment là, auquel cas chaque unité devra avoir le service **setenv@graphical.service** activé (comme indiqué précédemment, il ne suffit pas de le mettre en After dans le fichier `graphical.target` pour qu'il soit activé avant les autres).

Définir une session graphique comme service peut sembler appréciable. Deux problèmes se posent cependant : Le premier est que **systemctl** retourne immédiatement. Ainsi, un fichier `.xsession` contenant uniquement

```
# $HOME/.xsession
export DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
systemctl --user start graphical.target
```

ne sera pas suffisante, sous peine de voir sa session graphique disparaître à peine commencée (le `export DBUS_SESSION_BUS_ADDRESS` est nécessaire pour le dialogue avec **systemd**). Il existe des programmes tels que systemd-wait qui vont attendre la fin d'un service donné et peut être utilisé (selon le type de session).

Le deuxième problème est plus ennuyeux : il vient de la distinction entre la session utilisateur elle-même et les services. Si on ne fait pas attention, on risque de se retrouver avec quelque chose comme ça :

```
    user-1000.slice
        session-c1.scope
            systemd-wait -q --user graphique.service failed inactive
        user@1000.service
            systemd --user
            graphique.service
                openbox
                firefox
                xterm
                xterm
            service_2.service
            service_3.service
```

Cela peut être gênant, parce que essentiellement le seul programme ici qui est considéré comme “actif” est le programme **systemd-wait** qu'on a mis dans le `.xsession`.

slim n'est pas compatible avec systemd –user: **slim** est un gestionnaire de connexion, c'est-à-dire de façon simplifiée un équivalent du login dans les tty mais en graphique. Cela a plusieurs implications : il doit s'occuper de démarrer un serveur X, et de démarrer la session de l'utilisateur après la connexion. Mais il doit aussi reprendre la main quand l'utilisateur s'en va.
\\Au moment de la connexion de l'utilisateur (via la configuration dans pam), **slim** démarre une instance de `systemd --user`, ainsi qu'un script de l'utilisateur. Il ne démarre pas de serveur X puisqu'il est déjà démarré.

Au moment de la déconnexion cependant, il ne redémarre pas tout et réutilise l'état dans

lequel il était. Du point de vue de **systemd**, **slim** passe donc dans le “scope” de l'utilisateur précédent (même s'il est toujours lancé par root)

Ce comportement là est ennuyeux pour les sessions suivantes. Une solution est de mettre une ligne dans `/etc/slim.conf` de la forme:

```
sessionstop_cmd systemctl restart slim.service
```

qui aura pour effet de redémarrer slim à la fin de chaque session.

Polkit et les droits d'administration

Systemd distingue deux types d'utilisateur, à savoir les utilisateurs “actifs” et “inactifs”. Par exemple si deux utilisateurs sont connectés physiquement sur le même pc, un seul est réellement “actif”. Il fait également une distinction selon que l'utilisateur est local ou distant. L'information peut être obtenue avec

```
loginctl show-session "session"
```

où “session” est l'identifiant de la session (on peut lister les sessions avec `loginctl list-sessions`). En particulier, et c'est là que ça va devenir important, un service est toujours inactif.

Polkit est un service qui permet de donner, temporairement ou non, plus de droits à un utilisateur. Ces droits sont classés selon l'état de l'utilisateur, actif ou inactif ou autre.

Polkit et **systemd** n'ont pas la même notion d'utilisateur actif : pour **polkit**, un utilisateur actif/inactif est local et “actif/inactif au sens de systemd”. Tous les autres sont “any” (en particulier les utilisateurs distants). Cette règle est hardcodée dans la fonction **check_authorization_sync** du fichier `src/polkitbackend/polkitbackendinteractiveauthority.c` du code source de polkit.

Un **xterm** démarré en tant que service et un démarré directement (par exemple via le fichier `.xsession`) n'auront pas les mêmes droits du point de vue de polkit ! Il faut donc faire attention à cela lorsqu'on souhaite utiliser `systemd --user` comme gestionnaire de session, car par exemple il sera incapable d'éteindre le pc sans sudo.