Dr. Oliver Diedrich

Zwangsabmeldung

Session Management unter Linux

Wenn der Rechner den Anwender nach einer gewissen Zeit zwangsabmeldet, muss man nicht jeden Tag aufs Neue mit dem Nachwuchs über Nutzungszeiten diskutieren. Unter Linux erweist sich diese Aufgabe als erstaunlich komplex und erfordert einen Ausflug in die Tiefen von Session Management und D-Bus-Programmierung.

Eltern kennen das Problem: Der Nachwuchs soll nicht unbegrenzt Zeit am Rechner verbringen. Eine Vereinbarung, wie lange der Computer genutzt werden darf, ist schnell getroffen; problematischer ist die Umsetzung in die Praxis. Eine technische Lösung, die für einen Zwangs-Logout nach der festgelegten Zeit sorgt, erspart tägliche Diskussionen – und erweist sich auch bei Schulrechnern, Kiosk-Systemen und so weiter als nützlich.

Wer über Dinge wie ein automatisches Logout nach einer gewissen Zeit nachdenkt, landet schnell beim Session Management von Linux. Sobald sich ein Benutzer lokal an dem grafischen Login-Schirm anmeldet, startet der Display Manager – bei Ubuntu 11.10, der Basis unserer Experimente, ist das lightdm, sonst meist gdm (Gnome) oder kdm (KDE) – eine neue Session für diesen Benutzer. Alle Prozesse einer solchen Session gehören demselben Benutzer, haben einen gemeinsamen Vorfahren – den vom Display Manager gestarteten Session Manager (auch Session Leader genannt) – und teilen eine gemeinsame Umgebungsvariable.

In der Praxis umfasst die Session eines lokal angemeldeten Benutzers seinen Desktop und alle auf diesem Desktop gestarteten Anwendungen. Loggt sich ein bereits lokal angemeldeter Benutzer remote etwa per ssh ein, startet eine zweite Session für die via ssh gestarteten Prozesse. Die gemeinsame Umgebungsvariable der Sessions ist XDG_SESSION_COOKIE, der Session Manager als gemeinsamer Vorfahr ist beim Gnome-Desktop das Programm gnome-session, das der Display Manager beim Anmelden gestartet hat: Wenn dieser Prozess stirbt, endet die Session. Man könnte also einfach beim Einloggen einen Timer starten, der nach einer bestimmten Zeit den Prozess gnome-session per KILL-Signal abschießt.

Miteinander reden!

Das ist freilich die Holzhammermethode und nicht der Weg, wie die Dinge heutzutage auf dem Linux-Desktop laufen. Die Prozesse einer Desktop-Session haben nämlich eine weitere Gemeinsamkeit: Sie kommunizieren über eine Session-spezifische D-Bus-Instanz miteinander, über die man auch den Session Manager ansprechen kann [1]. D-Bus ist ein Protokoll zur Interprozess-Kommunikation, wobei auf Linux-Systemen typischerweise mindestens zwei Busse laufen: ein System-Bus für die Session-übergreifende Kommunikation zwischen Anwendungen und Systemkomponenten sowie jeweils ein Session-Bus für jede Session.

Prozesse können D-Bus-Objekte am System- oder Session-Bus anmelden und dort
- Methoden bereitstellen, die andere Prozesse über den D-Bus aufrufen können;
- Signale verschicken, auf die andere am D-Bus registrierte Objekte reagieren können;

– die Methoden anderer D-Bus-Objekte aufrufen;
– auf Signale anderer D-Bus-Objekte reagieren.

Der Textkasten unten beschreibt, wie man herausfindet, welche Objekte sich auf Session- und System-D-Bus registriert haben, welche Funktionen sie bereitstellen und welche Signale sie verschicken.

Der Session Manager von Gnome bietet eine Reihe von Methoden an, die sich aus einem Programm oder mit einem Tool wie qdbus über den D-Bus aufrufen lassen [2]. So bringt der Befehl

```
qdbus org.gnome.SessionManager ⤷
                    /org/gnome/SessionManager Logout 0
```

den üblichen Logout-Dialog auf den Schirm: Qdbus ruft die Methode Logout des Objekts /org/gnome/SessionManager des D-Bus-Dienstes org.gnome.SessionManager mit dem Argument 0 auf. Mit 1 aufgerufen verzichtet die Logout-Methode auf die Rückfrage beim Anwender, mit dem Argument 2 können auch Anwendungen das Abmelden nicht verhindern: Der Session Manager fragt normalerweise – natürlich per D-Bus – vor dem Beenden der Session bei allen laufenden Anwendungen nach, ob sie etwas gegen das Abmelden einzuwenden haben. Der Editor Gedit beispielsweise blockiert den Logout, solange es noch ungespeicherte Änderungen gibt.

So reicht ein simples Skript aus, um eine bereits recht flexible Zwangsabmeldung nach einer bestimmten Zeit am Rechner zu implementieren:

```
#!/bin/sh
sleep 7200
qdbus org.gnome.SessionManager ⤷
                    /org/gnome/SessionManager Logout 0
sleep 300
qdbus org.gnome.SessionManager ⤷
                    /org/gnome/SessionManager Logout 0
sleep 30
qdbus org.gnome.SessionManager ⤷
                    /org/gnome/SessionManager Logout 2
```

Das Skript wartet zwei Stunden und zeigt dann den Logout-Dialog an. Bricht der Anwender das Abmelden ab, weil er noch etwas Wichtiges zu tun hat, kriegt er fünf Minuten Gnadenfrist, dann erscheint der Dialog erneut. Eine halbe Minute später erfolgt dann die Zwangsabmeldung. Der Logout-Dialog mit eingebautem Timer und der Möglichkeit, den Logout abzubrechen, erspart einem die Mühe, selbst einen Hinweis auf die bevorstehende Abmeldung auf den Bildschirm zu bringen.

Wenn man nun noch in /home/FOO/.config/autostart eine Desktop-Datei logout.desktop für das Skript anlegt, startet es automatisch, sobald sich User FOO anmeldet:

```
[Desktop Entry]
Type=Application
Exec=/usr/local/bin/logout.sh
X-GNOME-Autostart-enabled=true
Name=Auto-Logout
```

Jetzt fehlt dem Skript nur noch ein bisschen Logik, um zu verhindern, dass sich der Benutzer nach dem Abmelden sofort wieder anmeldet – ein wie auch immer gearteter Timestamp, der zu Beginn des Skripts geprüft und gesetzt wird –, und vielleicht noch so etwas wie festgelegte Nutzungszeiten: Beim Anmelden vor 16:00 oder nach 22:00 erfolgt ein sofortiger Zwangs-Logout. Man kann das Skript auch um eine User-Verwaltung erweitern und in /etc/xdg/autostart installieren, dann wird es bei jedem Anwender ausgeführt – und ist dem unmittelbaren Einflussbereich der User entzogen.

Herr der Sessions

Leider gibt es einen Schönheitsfehler: Zwar kann man Skript und Autostart-Datei als root anlegen und so vor Manipulationen schützen, aber das Skript läuft mit der User-Kennung des angemeldeten Benutzers, sodass der es mit kill einfach abbrechen kann. Das Set-User-ID-Bit, das dafür sorgt, dass der Prozess mit der User-ID des Dateibesitzers (also root) läuft, ist keine Lösung: Aus Sicherheitsgründen ignoriert Linux bei Skripten das SUID-Bit. Zudem weigert sich org.gnome.SessionManager, mit Prozessen zu kommunizieren, die unter einer anderen User-ID laufen. Aber es gibt einen Ausweg.

ConsoleKit ist ein Systemdienst, der auf Linux-Desktops läuft, sich über den System-D-Bus ansprechen lässt und über die laufenden Sessions wacht. Jede neue Session meldet sich dort an, und ConsoleKit sorgt beispielsweise dafür, dass man das System nicht einfach herunterfahren kann, wenn noch Sessions anderer Benutzer laufen.

Den D-Bus erforschen

Aus Entwicklersicht ist der D-Bus vor allem ein API zum Zugriff auf zahlreiche Systemfunktionen. Mit Programmen wie Qdbus (aus dem Programmpaket libqt4-dbus) oder Mdbus2, die aktuellen Distributionen beiliegen, lässt sich dieses API erforschen. Ohne Argument aufgerufen, geben beide Tools eine Liste der Dienste aus, die sich am Session-Bus angemeldet haben. Mit der Option --system erhält man stattdessen die Dienste am System-Bus. Die Namen der Dienste folgen meist dem Schema „Hersteller.Dienst“, also etwa org.freedesktop.NetworkManager oder com.canonical.Unity.

Mit einem solchen Dienst als Argument aufgerufen, geben die Tools die Pfadnamen der Objekte aus, die der Dienst registriert hat. AccountService beispielsweise legt neben dem Objekt /org/freedesktop/Accounts für allgemeine Funktionen wie Anlegen und Löschen von Usern ein eigenes Objekt für jeden User im System an, über das man Passwort, Shell und zahlreiche weitere benutzerspezifische Informationen setzen kann.

Welche Methoden ein Objekt bereitstellt, welche Signale es verschickt und welche Eigenschaften (Properties) es hat, erfährt man bei Angabe von Dienstname und Objektpfad:

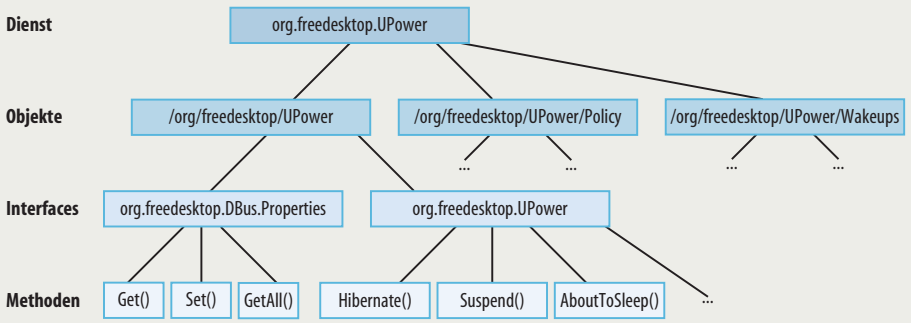
```
qdbus --system org.freedesktop.UDisks ⤷
                    /org/freedesktop/UDisks/devices/sda1
```

Die Methoden sind dabei über unterschiedliche Interfaces organisiert, die verwandte Funktionen zusammenfassen. So bietet fast jedes D-Bus-Objekt ein Interface mit dienstspezifischen Funktionen. Dazu können noch weitere Interfaces kommen: org.freedesktop.DBus.Properties etwa bietet Methoden zum Lesen und Setzen von Objekt-Eigenschaften; die Funktion Introspect() des Interface org.freedesktop.DBus.Introspectable nutzen Tools wie qdbus zum Abfragen der Objektfähigkeiten.

Um eine Methode aufzurufen, muss man qdbus und mdbus2 Dienstname, Objektpfad und Methodenname mit vorangestelltem Interface übergeben: Der Befehl

```
qdbus org.gnome.ScreenSaver / ⤷
                    org.gnome.ScreenSaver.Lock
```

ruft die Methode Lock des Interface org.gnome.ScreenSaver des Objekts / des ScreenSavers auf. Bei Qdbus ist die Angabe des Interface häufig nicht erforderlich, während Mdbus2 Methoden ohne Interface-Angabe nicht findet.



Dienste organisieren die angebotenen Funktionen über D-Bus-Objekte und Interfaces.

org.freedesktop.ConsoleKit registriert drei Objekte [3]: /org/freedesktop/ConsoleKit/Manager hält Methoden unter anderem zum Abfragen der laufenden Sessions und der genutzten Seats, zum Herunterfahren und Neustarten des Rechners und zum Anmelden und Abmelden von Sessions vor. Außerdem verschickt das Manager-Objekt Signale, wenn ein Seat angelegt oder entfernt wird. Zusätzlich gibt es Session-Objekte für jede laufende Session (/org/freedesktop/ConsoleKit/Session*n*) sowie Seat-Objekte für jeden Seat (/org/freedesktop/ConsoleKit/Seat*n*).

Ein Seat bezeichnet in der ConsoleManager-Terminologie in etwa eine physische Konsole samt den Sessions, die diese Konsole nutzen. Lokale Sessions laufen auf dem ersten Seat (den bedient zumeist der X11-Server, aber auch die virtuellen Konsolen gehören zum ersten Seat); für alle anderen Sessions werden eigene Seats angelegt. Das Fast User Switching („Benutzer wechseln“) moderner Linux-Desktops sowie Text-Logins auf den nicht vom X-Server belegten virtuellen Konsolen machen es möglich, mehrere Sessions auf dem ersten, lokalen Seat laufen zu lassen; allerdings kann immer nur eine Session auf einem Seat aktiv sein. Über das Seat-Objekt lassen sich die darauf laufenden Sessions abfragen; außerdem sendet es Signale aus, wenn Sessions hinzukommen oder enden.

Wenn man komplexere Aktionen auf dem D-Bus ausführen will, etwa auf Signale reagieren, reicht ein simples Tool wie Qdbus nicht mehr aus. Die D-Bus-Bibliothek selbst bietet nur Low-Level-Funktionen und ist nicht zur direkten Verwendung in Anwendungen gedacht – sie dient lediglich als Grundlage für verschiedene D-Bus-Bindings, unter anderem für Glib (C), Qt (C++) und das Python-Modul DBus-Python [4].

Mit Python auf den Bus

Letzteres macht den Umgang mit D-Bus recht bequem: Nach dem Import des Moduls stellt dbus.SystemBus() die Verbindung zum System-D-Bus her. Die Kommunikation mit dem Manager-Objekt von ConsoleKit erfolgt über ein Proxy-Objekt, im Beispiel ck_proxy, das die Methoden des Objekts bereitstellt – etwa GetSeats() zum Abfragen der Seats. DBus-Python bemüht sich dabei, die strikt festgelegten D-Bus-Datentypen auf passende Python-Objekte abzubilden: Aus dem „Array of ObjectPaths“ (ao), das GetSeats() zurückliefert, wird so einfach eine Liste.

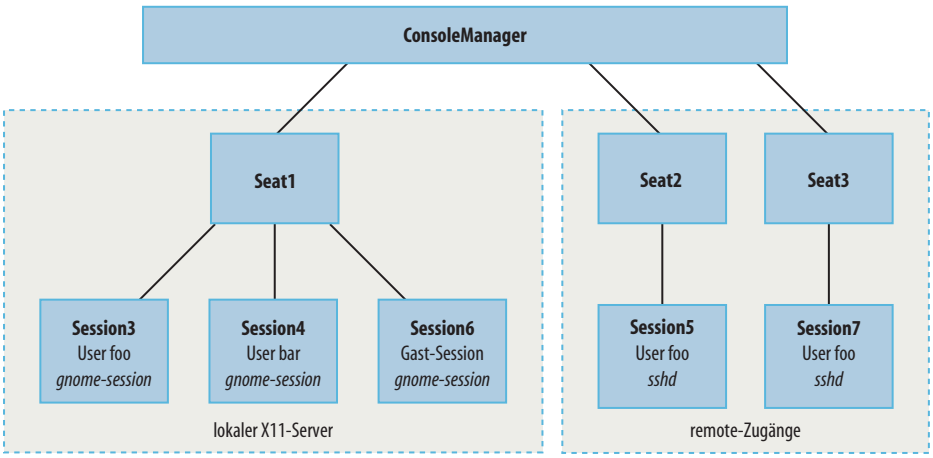
```
import dbus
bus = dbus.SystemBus()
ck_proxy = bus.get_object('org.freedesktop.ConsoleKit',
  '/org/freedesktop/ConsoleKit/Manager')
seats = ck_proxy.GetSeats()
```

Über die Liste der Seats kann man iterieren, für jedes Seat-Objekt ein Proxy-Objekt anlegen und darüber die Sessions auf dem Seat sowie die aktive Session abfragen:

```
for seat in seats:
  print 'Seat:', seat
  seat_proxy =
bus.get_object('org.freedesktop.ConsoleKit', seat)
  print 'Sessions:'
  for s in seat_proxy.GetSessions():
    print s
  try:   # fails when no active session exists
    print 'Active Session:',
    seat_proxy.GetActiveSession()
  except:
    print 'no active session'
  print
```

Die Methode GetSessions() des Manager-Objekts gibt die laufenden Sessions aus; über die zugehörigen Session-Objekte erfährt man Details über die Sessions:

```
for session in ck_proxy.GetSessions():
  proxy = bus.get_object('org.freedesktop.ConsoleKit',
  session)
  print 'Session:', session,
  if proxy.IsLocal() == True:
    print '(lokal)'
    print 'X11 Display:', proxy.GetX11Display(),
    print '(', proxy.GetX11DisplayDevice(), ')'
  else:
    print '(remote)'
    print 'Display Device:', proxy.GetDisplayDevice()
    print 'remote host name:',
    print proxy.GetRemoteHostName()
  user = proxy.GetUnixUser()
  print 'User ID:', user
```



Alle lokalen Sessions laufen auf dem ersten Seat.

Verdrahtet

Auch das Lauschen auf Signale macht DBus-Python recht einfach: Die Anweisungen

```
seat_proxy.connect_to_signal('SessionAdded',
session_added)
seat_proxy.connect_to_signal('SessionRemoved',
session_removed)
```

verbinden die Signale SessionAdded und SessionRemoved, die das Seat-Objekt beim Anlegen und Entfernen einer Session verschickt, mit den Funktionen session_added() und session_removed(). Die Signale übergeben den Funktionen den Pfad des Session-Objektes (beispielsweise /org/freedesktop/ConsoleKit/Session5):

```
def session_added(session):
  ses_proxy = bus.get_object('org.freedesktop.ConsoleKit',
  session)
  try:
    user = ses_proxy.GetUnixUser()
    local = ses_proxy.IsLocal()
  except:
    user = -1
  print 'Session', session, 'added'
  if user > -1:
    print 'User:', user,
    if local == True:
      print '(lokal)'
    else:
      print '(remote)'
def session_removed(session):
  print 'Session', session, 'removed'
```

Die Funktion session_added() legt ein Proxy-Objekt für die Session an, um einige Informationen über sie auszugeben. Die Ausnahmebehandlung beim Aufruf von GetUnixUser() und IsLocal() ist nötig, da beim Start einer Gast-Session zunächst eine temporäre Session angelegt und entfernt wird, bei der das Abfragen der Eigenschaften unter Umständen zu einer Exception in der D-Bus-Bibliothek führt. Man kann diese Funktion auch beim Iterieren über die schon bestehenden Sessions beim Start des Skripts aufrufen – der Funktion ist es egal, ob sie über ein Signal von ConsoleKit oder direkt aufgerufen wird.

Nun muss man noch auf das Signal SeatAdded des Manager-Objekts reagieren, das als Parameter den Objektpfad des neuen Seats mitbringt. Für den erzeugt man wie oben ein Proxy-Objekt, um die Signale SessionAdded und SessionRemoved mit den entsprechenden Funktionen zu verbinden:

```
def seat_added(seat):
  print 'Seat', seat, 'added'
  seat_proxy = bus.get_object('org.freedesktop.ConsoleKit',
  seat)
  for s in seat_proxy.GetSessions():
    session_added(s)
  seat_proxy.connect_to_signal('SessionAdded',
  session_added)
  seat_proxy.connect_to_signal('SessionRemoved',
  session_removed)
```

Für die Verdrahtung sorgt

```
ck_proxy.connect_to_signal('SeatAdded', seat_added)
```

nach dem Anlegen des Proxy-Objekts für das Manager-Objekt von ConsoleKit.

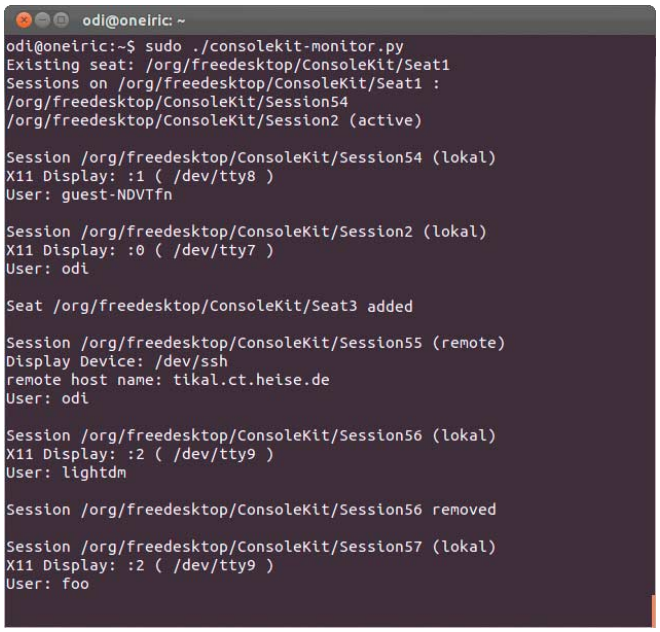
Um einlaufende Signale verarbeiten zu können, benötigt das Skript eine globale Event-Schleife. DBus-Python greift dazu auf Glib zurück:

```
from dbus.mainloop.glib import DBusGMainLoop
import gobject
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
```

Die Event-Schleife muss gesetzt werden, bevor man die Verbindung zum D-Bus herstellt. Nach dem Verdrahten der Signale mit den zuständigen Funktionen startet

```
loop = gobject.MainLoop()
loop.run()
```

die Schleife. Das Python-Skript gibt jetzt alle existierenden, neu angelegten und entfernten Seats und Sessions aus, bis es mit Strg+C beendet wird. Den kompletten ConsoleKit-Monitor können Sie als Listing über den c't-Link herunterladen.



Seat1 repräsentiert den lokalen PC mit zwei laufenden Sessions, Seat3 ein SSH-Login. Beim Fast User Switching startet zunächst eine lightdm-Session.

Unter Kontrolle

Nachdem jetzt sämtliche Sessions einmal die Funktion session_added() durchlaufen, ist auch klar, wo die Kontrolle der Login- und Nutzungszeiten erfolgen muss. Dazu benötigt man zunächst eine Datenstruktur, die für jeden User festlegt, zwischen welchen Uhrzeiten er sich einloggen und wie lange er angemeldet bleiben darf. Unser Autologout-Skript, das Sie über den c't-Link finden, legt dazu für jeden Anwender mit Zugangsrestriktionen ein Objekt der Klasse User an, das – mit der User-ID als Index – in dem globalen Array users gespeichert wird. Die Zugangsbeschränkungen konfiguriert man beim Initialisieren der User-Objekte. User-IDs ab 115 werden für Gast-Sessions verwendet – Ubuntu legt für jede parallel gestartete Gast-Session einen neuen User an. Wenn man hier das Limit auf „0:01“ setzt, beendet das Skript Gast-Sessions kurz nach dem Start.

Die Methode activate() des User-Objekts überprüft, ob ein Benutzer seine Limits erreicht hat. Sie wird unter anderem aus session_added () aufgerufen, wenn eine X11-Session beim Programmstart entdeckt wird oder eine neue X11-Session startet. Das Programm berücksichtigt nur lokale X11-Sessions, Text-Logins (ob lokal oder remote) bleiben außen vor – die muss man im Zweifelsfall abschalten. Jeder User kann nur eine lokale X11-Session haben, das vereinfacht die Verwaltung der Sessions.

Allerdings gibt es eine andere Schwierigkeit: Über das Fast User Switching können mehrere Benutzer gleichzeitig angemeldet sein; und natürlich soll die Viertelstunde, in der Papa „mal schnell“ seine E-Mail checkt, nicht von der erlaubten Zeit des Sohnemanns abgehen. Daher muss man im Blick behalten, ob die überwachten Sessions aktiv sind.

Das User-Objekt überprüft daher nicht nur die Limits beim Aktivieren einer Session, sondern protokolliert in den Variablen active_since und cumulated mit, seit wann die Session aktiv ist und wie lange der User insgesamt schon aktiv war. Diese Informationen müssen natürlich ein Ab- und erneutes Anmelden überstehen, daher berechnet die Methode deactivate() beim Abmelden, wie lange die Session in den letzten 24 Stunden aktiv war, und speichert diesen Wert in User.cumulated.

Damit die Zugangsbeschränkungen auch einen Neustart des Skripts (etwa durch einen Reboot des Rechners) überdauern, schreibt das Skript die User-Objekte mit Hilfe des pickle-Moduls bei jeder Zustandsänderung (etwa der Aktivierung oder Deaktivierung einer Session) in Dateien im Verzeichnis /usr/share/autologout; als Dateiname muss die User-ID herhalten. Wenn Sie Änderungen an der Konfiguration eines Users im Skript vornehmen, müssen Sie die zugehörige Datei löschen, sonst lädt das Skript das User-Objekt beim nächsten Start aus der Datei – mit den dort gespeicherten Einstellungen des letzten Programmlaufs.

Natürlich muss irgendwann ein Reset der kumulierten Nutzungszeit erfolgen, sonst würde sich ein Anwender, der einmal seine zwei Stunden Nutzungszeit verbraucht hat, nie mehr anmelden können. Dafür sorgt die Methode reset() des User-Objekts, die activate() vor dem Checken der Restriktionen aufruft, falls der letzte Reset (oder der Start des Skripts) länger als 24 Stunden zurückliegt.

Aktivitäten

Wenn sich der Aktivitätsstatus einer Session ändert, verschickt sie das Signal ActiveChanged mit einem Integer-Wert, der angibt, ob die Session aktiviert (1) oder deaktiviert (0) wurde. Der Signal-Handler erfährt allerdings nicht, von welcher Session das Signal stammt. Da zu jeder überwachten Session ein eigenes User-Objekt gehört, bietet es sich an, den Signal-Handler als Methode der User-Klasse zu implementieren – die weiß ja, für welche Session sie zuständig ist:

```
class User:
    ...
    def active_changed(self, is_active):
        if is_active:
            self.activate()
        else:
            self.deactivate()
```

Die Verknüpfung der Methode mit dem Signal ActiveChanged erfolgt in der Funktion session_added(). Allerdings wirft DBus-Python dabei eine KeyError-Exception, da der Integer-Wert des ActiveChanged-Signals nicht zu den Parametern self, is_active der Methode passt. Man kann die Exception aber einfach mit

```
try:
    ses_proxy.connect_to_signal('ActiveChanged',
    users[uid].active_changed)
except KeyError:
    pass
```

abfangen, dann funktioniert der Signal-Handler wie erwartet.

Beim Umgang mit dem Signal ActiveChanged sind ein paar Details zu beachten: Beim Starten einer Session wird das Signal nicht erzeugt, wohl aber beim Entfernen. Da die Methoden activate() und deactivate() des User-Objekts Statusinformationen des Objekts verändern, muss man selbst dafür sorgen, dass der SessionAdded-Handler User.activate() aufruft.

Tiefschlafphase

Auch Suspend und Hibernate erfordern eine spezielle Behandlung: Die im Schlafzustand verbrachte Zeit soll natürlich nicht in die Nutzungszeit einfließen. Wenn der Rechner einschlafen soll, verschickt UPower, zuständig für das Powermanagement, das Signal Sleeping über den D-Bus, beim Aufwachen Resuming:

```
upower_proxy = bus.get_object('org.freedesktop.UPower',
  '/org/freedesktop/UPower')
upower_proxy.connect_to_signal('Sleeping', sleeping)
upower_proxy.connect_to_signal('Resuming', resuming)
```

Die Signal-Handler müssen lediglich die aktive Session auf inaktiv beziehungsweise auf aktiv setzen – dafür reicht die Sekunde aus, die UPower Prozessen zum Verarbeiten des Sleeping-Signals zugesteht.

Bis einschließlich Version 0.9.3 hat UPower leider einen Fehler, der dazu führt, dass der Powermanager das Sleeping-Signal nicht zuverlässig verschickt – das betrifft zum Beispiel Ubuntu 10.04. Als Workaround könnte man bei Notebooks auf das Zuklappen des Deckels reagieren, was sich als Änderung der Property LidIsClosed auf True zeigt.

UPower sieht keine eigene Methode zum Abfragen dieser Variablen vor, daher muss man auf das Interface org.freedesktop.DBus.Properties zurückgreifen, das fast alle D-Bus-Objekte bieten und das man über die Funktion dbus.interface() auswählt. Da die Callback-Funktion für das Signal Changed, das UPower bei jeder Änderung einer seiner Eigenschaften verschickt, auf dieses Interface zugreifen muss, ist es im Skript als globale Variable deklariert:

```
iface = ''
...
def changed():
  if iface.Get('', 'LidIsClosed'):
    sleeping()
...
upower_proxy = bus.get_object('org.freedesktop.UPower',
  '/org/freedesktop/UPower')
iface = dbus.Interface(upower_proxy,
  'org.freedesktop.DBus.Properties')
version = iface.Get('', 'DaemonVersion')
maj, min, sub = version.split('.')
if int(min) == 9 and int(sub) < 4:  # UPower < 0.9.4
  upower_proxy.connect_to_signal('Changed', changed)
else:
  upower_proxy.connect_to_signal('Sleeping', sleeping)
```

Die Funktion Get() des Properties-Interface zum Abfragen von Objekteigenschaften erwartet als Parameter Interface- und Property-Name. Ersteres ist beim Aufruf über das Interface-Objekt ein leerer String (der aber nicht fehlen darf!), Letzteres die erfragte Eigenschaft LidIsClosed.

Terminator

Bleibt die Frage, wie das Skript eine Session beenden kann. ConsoleKit bietet zwar Methoden zum Starten des ScreenSavers (Lock() im Session-Objekt) und zum Herunterfahren des Rechners (Stop() im Manager-Objekt), aber keine Methode zum Aufruf des Logout-Dialogs oder zum Zwangsbeenden einer Session (die Methode CloseSession() darf lediglich der Prozess aufrufen, der die Session zuvor mit OpenSession() gestartet hat). Man muss also über den Session-Bus der zu beendenden Session mit dem Session Manager kommunizieren.

Aus Sicherheitsgründen ist D-Bus allerdings so konfiguriert, dass lediglich Prozesse, die der Session angehören, auf den Session-Bus zugreifen dürfen. Ein Prozess, der mit Root-Rechten lange vor dem ersten Login gestartet wird, um die Logins zu überwachen, gehört natürlich nicht dazu. Eine saubere Lösung für dieses Problem gibt es nicht; die hässliche Lösung besteht darin, die D-Bus-Policy so abzuändern, dass Root-Prozesse mit dem Session-Bus kommunizieren dürfen. Die Sicherheitsimplikationen erscheinen uns dabei aber vertretbar, schließlich dürfte ein Prozess mit Root-Rechten auch mit einem simplen killall gnome-session sämtliche Sessions abschießen – oder den Rechner durch Aufrufen der Stop()-Funktion von ConsoleKit einfach herunterfahren.

Den Root-Zugriff auf den Session-Bus erlaubt man durch Anlegen einer Datei /etc/dbus-1/session-local.conf mit folgendem Inhalt:

```
<busconfig>
  <policy context="default">
    <allow user="root" />
  </policy>
</busconfig>
```

Theoretisch erlaubt D-Bus auch feinere Rechteabstufungen, etwa das Eingrenzen der Erlaubnis auf bestimmte Sender, Empfänger und Methoden. Allerdings gerät man hier in einen Bereich, in dem D-Bus nur sehr rudimentär dokumentiert ist und wo die Dinge manchmal nicht so funktionieren, wie man es erwarten würde. Noch eine Warnung, falls Sie mit der D-Bus-Policy experimentieren wollen: Bei einem Fehler in session-local.conf verweigert der D-Bus-Daemon den Start und blockiert dabei auch den Start des Desktops. Abhilfe: Per ssh oder auf einem virtuellen Terminal einloggen und die Policy-Datei korrigieren.

Ab Version 0.81 kennt DBus-Python die Funktion dbus.bus.BusConnection(), die wie dbus.SessionBus() ein Bus-Objekt zurückliefert, das die Verbindung zu einem System- oder Session-Bus herstellt. Während dbus.SessionBus() mit dem Session-Bus der aktuellen Session verbindet (festgelegt über die Umgebungsvariable DBUS_SESSION_BUS_ADDRESS), kann man dbus.bus.BusConnection() die Adresse eines beliebigen Busses übergeben – man muss dazu nur dessen DBUS_SESSION_BUS_ADDRESS kennen.

Um diese Bus-Adresse zu erfahren, kann man die Prozessliste nach dem gnome-session-Prozess absuchen und DBUS_SESSION_BUS_ADDRESS aus /proc/*nnn*/environ auslesen – nicht elegant, aber es funktioniert:

```
for path in os.listdir('/proc/'):
  if os.path.exists('/proc/' + path + '/exe'):
    if 'gnome-session' in os.readlink('/proc/'+path+'/exe'):
      f = open('/proc/' + path + '/environ', 'rb')
      environ = f.read()
      env = environ.split('\0')
      for e in env:
        if '=' in e:
          var, val = e.split('=', 1)
        if 'DBUS_SESSION_BUS_ADDRESS' == var:
          address = val
      f.close()
      session_bus = dbus.bus.BusConnection(address)
```

session_bus ist das Proxy-Objekt für den Session-Bus der Gnome-Session, über das man an den Session-Manager herankommt:

```
session_manager =
session_bus.get_object('org.gnome.SessionManager',
'/org/gnome/SessionManager')
```

Über das dabei erzeugte Proxy-Objekt kann man wie üblich die Logout-Methode des Session-Managers aufrufen. Dabei ist eine D-Bus-Spezialität zu beachten: Variablen sind strikt typisiert und der Logout-Aufruf verlangt einen vorzeichenlosen 32-Bit-Integer (D-Bus-Datentyp ‚u'). Da die automatische Typkonvertierung von DBus-Python hier versagt, muss man selbst den passenden Datentyp erzeugen:

```
i = dbus.UInt32(0)
session_manager.Logout(i)
```

Und der Rest

Damit sind die wesentlichen Bausteine zusammen: Wann immer eine neue Session startet oder eine alte aktiviert wird, prüft User.activate(), ob die Nutzungsbeschränkung zuschlagen muss. Wenn ja, wird die Funktion terminate() direkt aufgerufen; wenn nein, sorgt die Timer-Funktion des (sowieso eingebundenen) gobject-Moduls für den rechtzeitigen Aufruf von terminate():

```
gobject.timeout_add_seconds(limit, terminate)
```

In unserem Autologout-Skript ist dem Aufruf von terminate() noch die Methode User.check_terminate() vorgeschaltet, die prüft, ob das Limit tatsächlich erreicht ist: Es könnte ja sein, dass die Session zwischen dem Setzen des Timers und seinem Ablauf für längere Zeit deaktiviert oder im Suspend war. In diesem Fall setzt User.check_terminate() einfach einen neuen Timer, statt die Session über einen Aufruf von terminate() direkt zu beenden.

Die Funktion terminate() wartet zunächst einige Sekunden, bis die Session sicher gestartet ist, bestimmt dann die Adresse des richtigen Session-Bus und startet erst einen abbrechbaren Logout, dann fünf Minuten später den harten Zwangs-Logout. Die D-Bus-Aktionen müssen dabei in einen try-Block eingeschlossen sein, damit keine DBusException auftritt, falls sich der Anwender in der Zwischenzeit selbst abgemeldet hat und der Session-D-Bus daher gar nicht mehr existiert. Die Funktion muss False zurückliefern, damit sich der Timer beendet – ansonsten wird sie periodisch immer wieder aufgerufen.

Das Autologout-Skript, das Sie über den c't-Link herunterladen können, muss mit Root-Rechten laufen und ist dafür gedacht, beim Systemstart anzulaufen. Es überwacht die Aktivitäten aller User, für die ein User-Objekt mit Beschränkungen angelegt wurde. Festlegen lassen sich Beginn und Ende des erlaubten Nutzungsintervalls (Parameter start und end) sowie die maximale Nutzung pro Tag (limit). (odi)

Literatur

[1] Natanael Mignon, Linux-Omnibus, D-Bus: Plug'n' Play auf dem Linux-Desktop, c't 23/06, S. 208.

[2] Gnome Session Manager: http://people.gnome.org/~mccann/gnome-session/docs/gnome-session.html

[3] ConsoleKit: http://www.freedesktop.org/software/ConsoleKit/doc/ConsoleKit.html

[4] D-Bus mit Python ansprechen: http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html

www.ct.de/1204164 c't