Your app everywhere, just in a Snap!

Daemon Snapper's Workshop

How to make Snaps from System Daemons and Utilities

Till Kamppeter, Sergio Cazzolato

CANONICAL ubuntu

Introduction

What you need to know:

- Basic installing from source, little code tweaks, …
- Basic snapping, for example you should have attended one of:
  - Snapping like Hell(sworth)
  - ROS Deployment Workshop

Daemon (system service)

- Runs permanently, from boot/install to shutdown/uninstall
- No direct user interface (sometimes web admin interface)
- No terminal interaction
- Listens for events: Network, D-Bus, Domain socket, UDEV, …
- Acts in response to events

Introduction

Daemons are not interactive, but this workshop is!

- We have a **little daemon** for you to try out everything: **`exampled`**
- It is a **simple C program** listening for raw data at a given port and storing it in a file
- You will **make a Snap** of this little daemon
- You will learn all aspects of making it working **under the confinements of a Snap**
- We will show **different methods** to fulfill the needed requirements
- We will apply advanced methods for using **UDEV** or for letting our daemon only **accept connections from Snaps which plug a given interface**
- We will answer your **questions** and **discuss** your needs

Contents

Hands-on exercises

This is a **workshop**, interactive, so you will try everything out on **your laptop**

- Have a **suitable Linux distribution** running on your laptop, **ideally Ubuntu**
- Have **snapd** and **snapcraft** installed
- Get **the example** from the following GitHub repository:

  https://github.com/sergiocazzolato/daemon-snapper.git

- During the workshop you will edit the files (mainly snapcraft.yaml) and build the Snap several times.

Daemon Snapper's Workshop

Defining and controlling a daemon in snapcraft.yaml

Treat an app as a daemon

- Make sure your app is actually a daemon (running permanently)
- Define it as a daemon in `snapcraft.yaml`, under `apps:`:

```
Apps:
  exampled:
    command: bin/exampled-run.sh
    daemon: simple
```

The `daemon:` entry has following modes available:

- `simple`: Daemon runs in foreground
- `forking`: Daemon forks background process and exits
- `oneshot`: System utility to run once on every boot, no daemon
- `notify`: Daemon notifies systemd

Treat an app as a daemon

- For most modern daemons we use **`daemon: simple`**, daemons which are forking by default we make **non-forking via command line**: `cupsd -f`
- We usually start the actual daemon through a **wrapper script** (here `exampled-run.sh`) to use env variables and command line options
- There are further **parameters for controlling the daemon**, also to be used in its **`apps:`** entry:
  - **`reload-command:`**, **`stop-command:`**, **`post-stop-command:`**: Additional scripts to handle restart, shutdown and clean-up
  - **start-timeout:**, **stop-timeout:**, **`watchdog-timeout:`**: Timeouts for reporting failure or killing a stuck daemon
  - **`before:`**, **`after:`**, **`timer:`**, … many more, see documentation: https://snapcraft.io/docs/services-and-daemons

Daemon Snapper's Workshop

Handling directories and files the daemon uses

Directories and files

- Daemons use several **directories and files**:
  - Configuration files
  - Log files
  - Data files
  - Spool directories
  - Domain sockets
  - …
- These are in **`/etc/`**…, **`/var/`**…, or **`/run/`**… by default, places not accessible from a Snap
- Need to be all under **`/var/snap/NAME/`**…/…

Directories and files

- Methods to **change the locations** (not necessarily available in all daemons)
  - **Command line options**
    - Via wrapper script
    - Example: **`exampled -l $SNAP_COMMON/log.txt`**
  - **Environment variables**
    - Via `snapcraft.yaml`, in `apps:`

```
apps:
  exampled:
    environment:
      LOGFILE: $SNAP_COMMON/log.txt
```

- Via wrapper script

Directories and files

- Methods to **change the locations** (not necessarily available in all daemons)
  - **Configuration file**
    - Provide modified/patched config file
    - Let wrapper script modify config file to assure, even if user edits
    - Config file must be under `/var/snap/NAME/…` , not under `/snap/NAME/…` or `/etc/…` (use the other methods described here for the config file location)
  - **Modifying upstream code**
    - Only open source, last mean
    - As patch for your Snap only
    - Actually upstream
      - You are upstream maintainer
      - Pull/Merge request
      - Must work in Snap and in classic installation
        - `if ((loc = getenv("SNAP_COMMON")) == NULL) loc = "/var";`

Daemon Snapper's Workshop

Controlling the snapped daemon

Controlling the snapped daemon

Snaps manage their own services without the need for manual intervention. However, snapd offers a set of commands to allow a snap's services to be inspected and their statuses changed

These are the commands provided:

Use snap services to lists all the services added to the system by the currently installed and enabled snaps

```
> $ snap services
```

Adding a snap name as an argument will list only those services added by that snap

```
> $ snap services daemon-example
```

Controlling the snapped daemon

Services are restarted using the snap restart <snap name> command. This may be necessary if you've made custom changes to the snap application

```
    > $ sudo snap restart daemon-example
```

The start and stop commands control whether a service should be currently running

```
    > $ sudo snap stop daemon-example.exampled
    > $ sudo snap start daemon-example.exampled
```

If you need to see the log output for a snap's services, use the logs command

```
    > $ sudo snap logs daemon-example
    > $ sudo snap logs daemon-example.exampled
```

Daemon Snapper's Workshop

System users and groups

System users and groups

- Daemons will **always be started as root** (standard apps as the calling user)
  - Snapped daemon started by system boot or Snap install
- Daemon (or wrapper script) could **drop privileges to a system user**
- snapd does **not allow use of the host system's users**, like `adm` or `lp`, and does not support adding system users and groups.
- Snapd has a user and a group, both named **`snap_daemon`** for this purpose
- To **activate them**, add to `snapcraft.yaml` (top level, no section):

```
system-usernames:
  snap_daemon: shared
```

- You will need to **configure your daemon** to use `snap_daemon` and not its own user/group: Config file, build options, patch, …
- Documentation: https://snapcraft.io/docs/system-usernames

Daemon Snapper's Workshop

Managing upstream code patches

Managing upstream code patches

- Why do we need to **modify (patch) the upstream code**?
    - Daemons and system software can often get **tricky to snap**
    - **Upstream's design** did not take Snap into account
    - Upstream **does not always accept the changes**, or it takes time until the next release
    - Even **classic Debian/RPM** packaging often needs patches
- **Examples** for patching needs
    - Directory and file **locations used are hard-coded**, no options, config, …
    - Daemon has to determine **what the client plugs**, to accept/deny inquiry
    - Daemon does **operations which are not allowed under confinement** (like `chown`/`chmod`) but are also not needed under confinement

Managing upstream code patches

In `snapcraft.yaml` (only relevant lines, patch is in `snap/local/` of your project repository):

```
parts:
  exampled:
    override-build: |
      patch -p1 < $SNAPCRAFT_PROJECT_DIR/snap/local/log.patch
      snapcraftctl build
```

Daemon Snapper's Workshop

Daemon dependencies – 2 daemons in 1 Snap

Daemon dependencies – 2 daemons in 1 Snap

Method 1: Sequencing directives in `apps:` entries

- **`before`**: Supplies an ordered list of apps which are only started when our daemon is up and running
- **`after`**: Supplies an ordered list of apps which must have been started before we start

In `snapcraft.yaml` it looks like:

```
apps:
  exampleauxd:
    daemon: simple
    after: [exampled]
```

Daemon dependencies – 2 daemons in 1 Snap

Method 2: In wrapper script check whether the other daemon is ready (wait if not):

- When systemd **reports other daemon ready but it is not** the case (Usually fault of that daemon)
- When **we are not a daemon**, but need a daemon of our Snap running (we cannot use sequencing directives then)

Wrapper script **checks in a loop** until timeout:

- Runs **status utility** of other daemon
- **Tries to access** other daemon
- Checks presence of **daemon's process** …

Daemon Snapper's Workshop

Daemon dependencies – External daemon

Daemon dependencies – External daemons

- Sequencing directives before: and after: only work on apps in the same Snap
- So for a dependency on an external daemon (other Snap or classically installed), method (2) of the previous section has to be used.
- The installation of a Snap can trigger the installation of another Snap (containing the needed daemon)
    - **Placeholder content interface** with needed Snap as **`default-provider`**:

```
plugs:
  foo-install-example:
    interface: content
    content: foo
    default-provider: example
    target: $SNAP_DATA/foo
```

Daemon dependencies – External daemons

- Placeholder content interface is a **dirty workaround**
  - Snap has **no explicit support for package dependencies** (Snap A needs Snap B)
  - Using this in **seeded Snaps** breaks the installation of the OS (bug)
  - Needs **native solution** in snapd
- The installation of a Snap cannot trigger the installation of a DEB or RPM package.

Daemon Snapper's Workshop

UDEV Rules – What?! snapd does not support them. But …

UDEV Rules

- UDEV rules are needed for daemons or utilities triggered by **appearing or disappearing** of certain hardware
- Snap **does not support** supplying UDEV rules
- But a Snap plugging **`hardware-observe`** has enough access to observe hardware appearing/disappearing with the **`udevadm`** utility -> **workaround**
- Wrapper script runs **`udevadm monitor`** to observe hardware
- It parses the output lines and **filters them for the relevant hardware**
- It **starts/stops the actual daemon** (for converting protocols, …) or **triggers utilities** (for mounting files systems, loading firmware, creating print queue, …)
- Independent of whether actual app is daemon, **the script** with `udevadm monitor` permanently running **is a daemon** -> `daemon: simple`
- **Good News: On actual UDEV rule support by snapd is worked on**

UDEV Rules

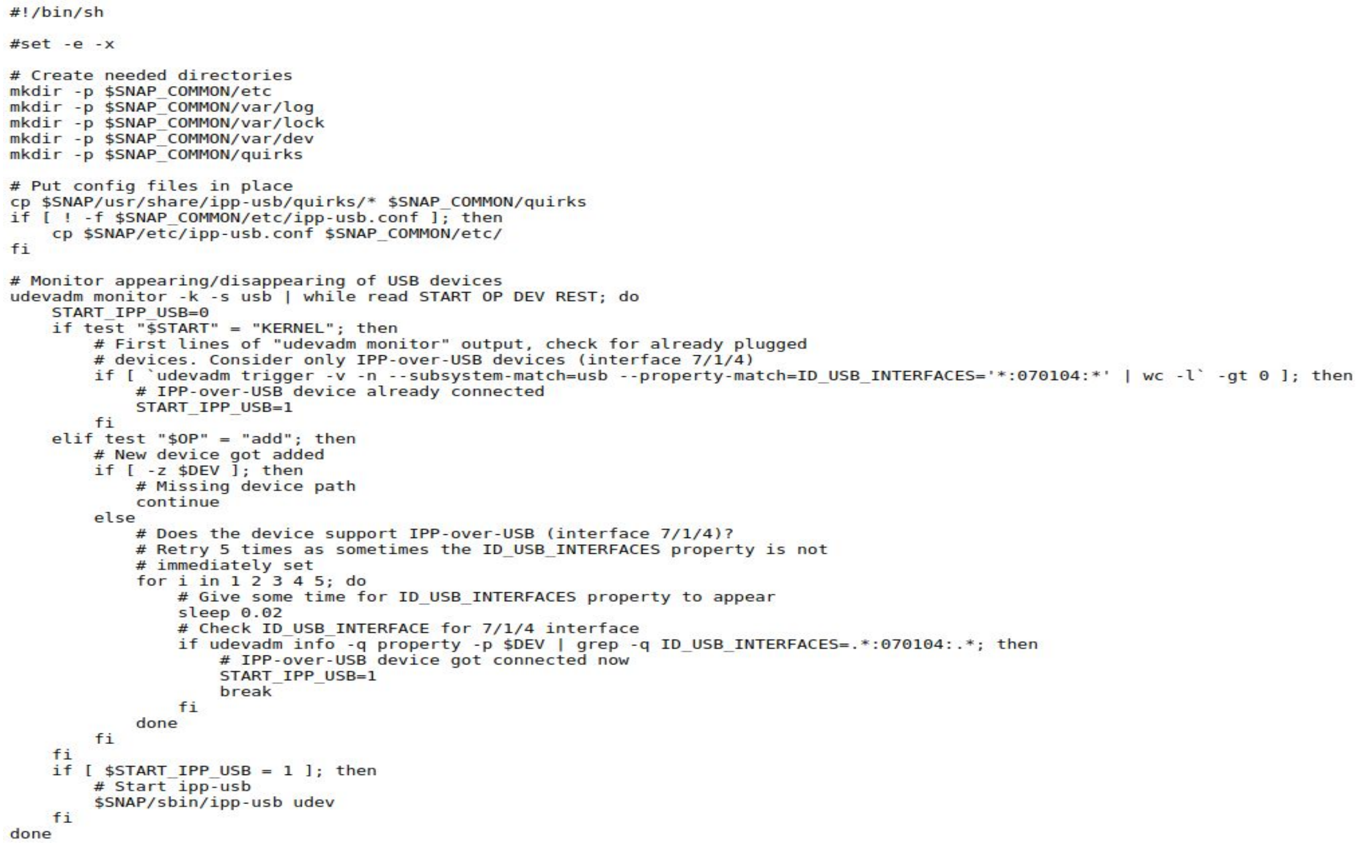
Example: ipp-usb: https://github.com/OpenPrinting/ipp-usb/

- Daemon for IPP-over-USB: Printer connected via USB to be accessed as network printer (IPP, Internet Printing Protocol) on localhost
- Printer is USB device with USB protocol 7/1/4
- ipp-usb comes with UDEV rule for classic installation
- Plugging the printer triggers start of ipp-usb daemon, daemon stops by itself on unplug, does not need to get triggered again by UDEV

UDEV Rules

- Check for already plugged printer:
  ```
  udevadm trigger --verbose --dry-run --subsystem-match=usb
    --property-match=ID_USB_INTERFACES='*:070104:*'
  ```
- Observe USB devices appearing/disappearing:
  ```
  udevadm monitor --kernel --subsystem-match=usb
  ```
- Filter 7/1/4 printer:
  ```
  udevadm info --query property --path $DEV |
    grep -q ID_USB_INTERFACES=.*:070104:.*
  ```
- Script is `snap/local/run-ipp-usb-server`in the ipp-usb repository

UDEV Rules



```
#!/bin/sh

#set -e -x

# Create needed directories
mkdir -p $SNAP_COMMON/etc
mkdir -p $SNAP_COMMON/var/log
mkdir -p $SNAP_COMMON/var/lock
mkdir -p $SNAP_COMMON/var/dev
mkdir -p $SNAP_COMMON/quirks

# Put config files in place
cp $SNAP/usr/share/ipp-usb/quirks/* $SNAP_COMMON/quirks
if [ ! -f $SNAP_COMMON/etc/ipp-usb.conf ]; then
    cp $SNAP/etc/ipp-usb.conf $SNAP_COMMON/etc/
fi

# Monitor appearing/disappearing of USB devices
udevadm monitor -k -s usb | while read START OP DEV REST; do
    START_IPP_USB=0
    if test "$START" = "KERNEL"; then
        # First lines of "udevadm monitor" output, check for already plugged
        # devices. Consider only IPP-over-USB devices (interface 7/1/4)
        if [ `udevadm trigger -v -n --subsystem-match=usb --property-match=ID_USB_INTERFACES='*:070104:*' | wc -l` -gt 0 ]; then
            # IPP-over-USB device already connected
            START_IPP_USB=1
        fi
    elif test "$OP" = "add"; then
        # New device got added
        if [ -z $DEV ]; then
            # Missing device path
            continue
        else
            # Does the device support IPP-over-USB (interface 7/1/4)?
            # Retry 5 times as sometimes the ID_USB_INTERFACES property is not
            # immediately set
            for i in 1 2 3 4 5; do
                # Give some time for ID_USB_INTERFACES property to appear
                sleep 0.02
                # Check ID_USB_INTERFACE for 7/1/4 interface
                if udevadm info -q property -p $DEV | grep -q ID_USB_INTERFACES=.*:070104:.*; then
                    # IPP-over-USB device got connected now
                    START_IPP_USB=1
                    break
                fi
            done
        fi
    fi
    if [ $START_IPP_USB = 1 ]; then
        # Start ipp-usb
        $SNAP/sbin/ipp-usb udev
    fi
done
```

Daemon Snapper's Workshop

Timers – Run a daemon only at certain times

Timers – Run a daemon only at certain times

When creating `snapcraft.yaml` to build a new snap, a snap's app could contain a `timer:` property. Timer strings are used in both refresh.timer setting and timer services as the timer.

A timer string is composed of one or more event sets, which are combined by using commas (, ,) as separators.

Each event set defines the weekdays and the time windows in which events may occur.

If no weekdays are provided, the default is every day. If no time windows are provided, the default is an arbitrary time in the day.

Timers – Run a daemon only at certain times

Some examples:

- `00:00-24:00/24` -> Every hour on the hour
- `00:00-24:00/48` -> Every 30 minutes
- `12:00-13:00/12` -> Every 5 minutes from 12:00 to 13:00
- `23:00` -> Every day at 23:00
- `Mon,10:00,,Fri,15:10` -> Mondays at 10:00, Fridays at 15:10
- `Mon,9:00~11:00,,Wed,22:00~23:00` -> Mondays, some time between 9:00 and 11:00, Wednesday, some time between 22:00 and 23:00
- `Mon,Wed` -> Monday and on Wednesday, at 0:00

Timers – Run a daemon only at certain times

Having this timer: `Mon,Fri,10:00,15:00`

Assuming today is Sunday, the next 5 events are, in order:

- `Monday 10:00`
- `Monday 15:00`
- `Friday 10:00`
- `Friday 15:00`
- `Monday 10:00`

The next event will be scheduled inside the soonest opportunity that matches both one of the provided weekdays and one of the provided time windows.

Daemon Snapper's Workshop

Snap Mediation – Allow connection only for client Snaps plugging interface XYZ

Snap Mediation – Allow connection only for client Snaps plugging interface XYZ

- **Really advanced topic: Requires changes in upstream code!**
- **Interfaces** give Snaps **controlled connection** to the outside world, like via
  - Additional, more **permissive AppArmor rules**
  - **Mounting** certain file systems
  - …
- A program, usually a daemon, **can determine** whether a client is a Snap and **which interfaces it is plugging**, so an interface can also
  - **Give permission to a certain service which requires it**
- This method of allowing access only to a Snap which plugs a certain interface is called **Snap Mediation**.
- AFAIK Snap Mediation is used by **CUPS** and **PulseAudio**

Snap Mediation – Allow connection only for client Snaps plugging interface XYZ

Example: The `cups-control` interface

- A Snap plugging **`cups-control`** has **full access to CUPS**, not only printing but also administrative tasks, like creating queues, or deleting anyone's jobs
- Therefore **`cups-control`** is a **"dangerous"** interface, needs explicit permission for auto-connection
- So we have also a **"safe"** interface named **`cups`** only for printing, which auto-connects
- But **how to stop admin tasks** when only **`cups`** is plugged?
- What **understands IPP** (Internet Printing Protocol) and knows what are administrative tasks is **solely CUPS**, resembling this in snapd would be a nightmare …

Snap Mediation – Allow connection only for client Snaps plugging interface XYZ

- So I have **modified the CUPS daemon** to check
  - Inquiry is administrative? No -> Allow, Yes -> Continue
  - Is client a confined Snap? No -> Allow, Yes -> Continue
  - Does client plug cups-control? No-> Deny, Yes -> Allow
- If **client is confined Snap**, AppArmor context is "`snap.NAME.` …", to be determined with **libapparmor**
- **Interfaces** which the client is plugging can be determined with **libsnapd-glib**
- See source code of CUPS, **`scheduler/auth.c`**.

Snap Mediation – Allow connection only for client Snaps plugging interface XYZ

- Note that the Snap Mediation is only in **CUPS 2.4.x or later**
- The **CUPS Snap** is only available **with Snap Mediation** (sufficiently new CUPS)
- To assure that a **Snap plugging `cups`** does not hit an **old, unprotected CUPS daemon** and could ~~manage it~~ mess it up, we do as follows:
    - The `cups` interface only connects to the **domain socket of the CUPS Snap** (which has Snap Mediation)
    - In the future (Ubuntu 23.04 or all-Snap distro) the CUPS Snap will be the actually used printing system
    - If the actually used printing system is a **classically installed CUPS**, the CUPS Snap's CUPS works as **proxy/firewall**, blocks admin requests and passes on print jobs

⇒ ⇒ ⇒ **So be careful when introducing Snap Mediation, there can be old versions of your target daemon still around!!**

Questions, Discussion, Links, and more ...

- First of all, for further info and more detail – **The Documentation**:
  - https://snapcraft.io/docs
  - https://snapcraft.io/docs/services-and-daemons
  - https://snapcraft.io/docs/system-usernames
- More questions and discussion – **The Forum**:
  - https://forum.snapcraft.io/
- More hands-on experience with our snappers – **The tutorial series going on**:
  - **Deploying Flutter apps on Linux Desktop (including snapping them)**
    Right after lunch, 14:00 -15:30, Palmkova 1 (This room)
  - **Snapping Desktop Applications (GNOME/GTK and KDE/Qt)**
    Right after lunch, 14:00 -15:30, Palmkova 2 (Room next to us)