Low level userspace stuff in python:

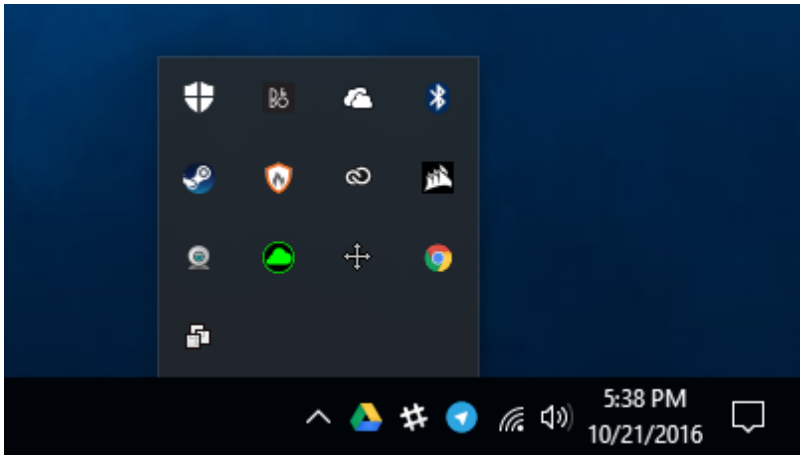
It’s not that complicated!

Code: github.com/ekatsah/fosdemx

Notes

- All this talk is about Internet of Things (IoT) firmware
- We are using python here, but really you can do that in any langage with a rich enough stdlib
- This is a simplification, we'll skip over some details, but the reality is not more complicated
- Code: github.com/ekatsah/fosdemx

Desktop, smartphone: process to abstract complex hardware to user => expose informations, user take decisions



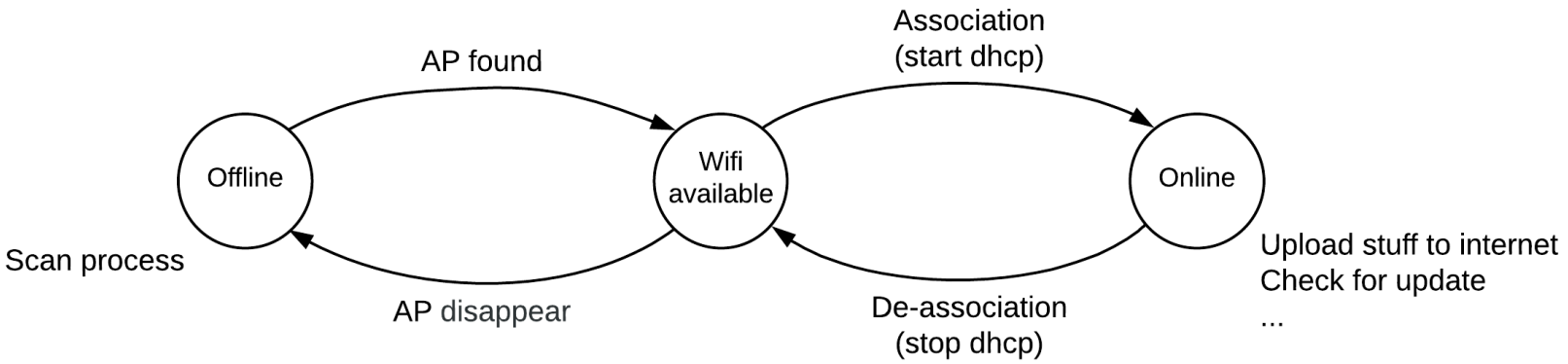
Autonomous systems?

- General architecture: has inputs, process it, take actions and send data to the mothership (cloud or whatever)
- Need to control stuff: it has a state
- For example, iot for car: different behavior power on/off, moving/idle, online/offline

State strategies

1) Nominal state and deal (ignore) with errors

2) Finite state machines



Depending of your case, but FSM are not really complicated and very robust, if your architecture is designed around this

Agenda

This presentation will not be about the reactive engine you should have in the processing pipeline. It’s about everything else: setup, init, process supervision, teardown, logs, monitoring, hotplugging...

Code

- The code for this system is here: https://github.com/ekatsah/fosdemx
- The same code, but ready to compile with buildroot: https://github.com/ekatsah/rpi-buildroot
- Example: mobile wifi scanner
  - Log every bssid and ssid of wifi with a gps position
  - Gps position will be faked

Low level setup

- Kernel expects to know what to call after it initialized
- Cmdline: `init=/bin/your_init` or in compile time
- Can be a script or binary

Low level setup: what to do

- Make directories and mount runtime filesystem (like /proc, /run, /sys, /dev/pts...)
- Load modules
- Easy to add configuration for debugging, for example set ip to eth0 etc

Usual filesystem structure

Mount point	Type	Notes
/	Ext2+, squashfs, yaffs2	Rootfs: should be read only
/var	Ext4, yaffs2	Data partition
/proc	procfs	Process description
/dev	devtmpfs	Device access
/dev/pts	devpts	Pseudo terminal access
/sys	sysfs	Kernel object access
/run et /var/run	tmpfs	Pid files, locks..
/tmp et /var/tmp	tmpfs	Temporary files.

Low level setup

```
from ctypes import CDLL, get_errno, c_char_p

def mount(source, target, fs):
    libc = CDLL("libc.so.0", use_errno=True)
    r = libc.mount(c_char_p(source.encode()), c_char_p(target.encode()),
                   c_char_p(fs.encode()), 0, 0)
    # check r for errors

mount("proc", "/proc", "proc")
# on raspi, no need to mount /dev
# mount("dev", "/dev", "devtmpfs")
make_directory("/dev/pts")
mount("devpts", "/dev/pts", "devpts")
mount("sys", "/sys", "sysfs")
mount("/dev/DATA", "/var", "ext4")
make_directory("/var/run")
mount("tmpfs", "/var/run", "tmpfs")
mount("tmpfs", "/var/tmp", "tmpfs")
symlink("/var/run", "/run")
symlink("/var/tmp", "/tmp")
```

https://github.com/ekatsah/fosdemx/blob/master/stack.py#L110

Chain scripts

- When the kernel called the low level board setup, this process is pid1
- If you want to use separate scripts for different part of the setup you should use `exec()`, not `fork()` nor `system()`

The zombie reaping responsability

- A zombie is a state of a process, when the `exit()` is called but the parent hasn't yet waited for its return code
- A process receive `SIGCHLD` when the kernel want it to reap a subprocess
- One of the role of pid1 is to reap the zombies

Process reaping: code

```
from os import waitpid, WNOHANG
from signal import SIGCHLD, signal



def reap_process(signum, frame):
    try:
        waitpid(-1, WNOHANG)
    except Exception as e:
        logger.warning("reap failed")


# in main...
signal(SIGCHLD, reap_process)
```

https://github.com/ekatsah/fosdemx/blob/master/stack.py#L31

Launch subprocess: strategies

- `System()`
- **`Popen()`**
- Custom spawn/daemonize: full blown `fork()` and `exec()`

Launch subprocess: code

```
class Process(object):
    def __init__(self, cmd):
        self.cmd = cmd
        self.name = self.cmd.split(" ")[0]
        self.process = None
        self._start()

    def _start(self):
        logger.info("start %s", self.cmd)
        self.process = Popen(self.cmd.split(" "), stdin=DEVNULL)

    def stop(self):
        logger.info("stop %s", self.name)
        if self.process is None:
            # not startable/stopped process, nothing to do
            return

        if self.process.poll() is None:
            self.process.terminate()
            self.process = None

    def check(self):
        if self.process is None:
            # not startable/stopped process, nothing to do
            return

        if self.process.poll() is not None:
            # dead process
            self.process = None
            self._start()
```

Missing:

- Stats (duration, restart count)
- Cooldown/backoff time
- Conditionnal input redirect
- Popen error management

https://github.com/ekatsah/fosdemx/blob/master/sv.py#L21

Launch subprocess: code

```
class SV(object):
    def __init__(self):
        self.processes = []

    def start_process(self, cmd):
        self.processes.append(Process(cmd))

    def stop_process(self, pattern):
        to_stop = [p for p in self.processes if match(pattern, p.cmd)]
        self.processes = [p for p in self.processes if not match(pattern, p.cmd)]
        for p in reversed(to_stop):
            p.stop()

    def stop_all(self):
        for p in reversed(self.processes):
            p.stop()
        self.processes = []

    def check(self):
        # restart dead process
        for p in self.processes:
            p.check()
```

https://github.com/ekatsah/fosdemx/blob/master/sv.py#L116

Launch subprocess: what to launch?

- Maybe you need vpn, sshd, ... => system apps
- If third party apps, don’t forget logging (syslogd, klogd..)
- What about time? NTP?
- Your apps

Launch subprocess: code

```
logger.info("starting logger, ssh, ntp, wifi")
sv.start_process("klogd -n")
sv.start_process("syslogd -n")
sv.start_process("dropbear -Fr /tmp/dropbear_key")
sv.start_process("ntpd -gnc /etc/ntp.conf")
sv.start_process("wpa_supplicant -iwlan0 -c/etc/wpa_supplicant.conf")
sv.start_process("getty 115200 /dev/tty1", redirect_input=False)

# do other stuff

# start apps
logger.info("start hilevel apps")
sv.start_process("diagnostic.py")
sv.start_process("fakegps.py")
```

https://github.com/ekatsah/fosdemx/blob/master/stack.py#L154

Example: Wifi monitoring

- `wifi_scan.py`: every 5s, scan the wifi with `wpa_cli scan_results`
- `fakegps.py`: every 1s, generate a new position
- `wifi_analyzer.py`: merge the two preceding streams

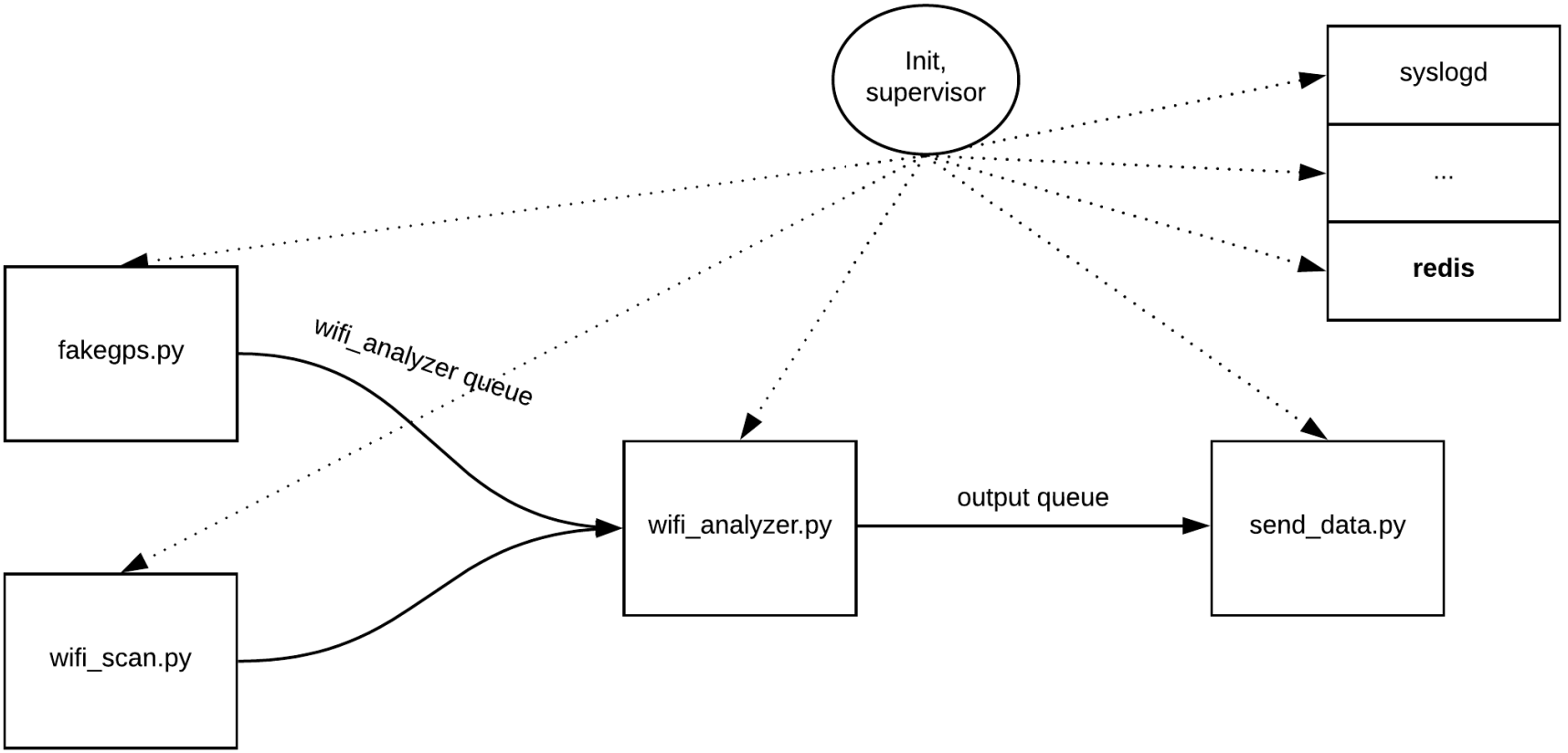
Inter Process Communication (IPC)

- Posix low level stuff: signal, fifo, shared memory, mq, filesystem
- Python datastructure: scary failure mode
- Third party system: sqlite, mq, redis?

IPC: how to choose?

- Simplicity of API
- Richness of API (Type system? Transaction?)
- Easyness to test/mock

IPC: why not redis?



The “Internet” of IoT

- We need network management
- We need to send stuff: mqtt/amqp..
- `pyroute2` lib
- FSM is really helpful there

Logging & monitoring

- Things will fail
- You need to understand what happened
- Make a small app to log every minute or so the state of memory, sdcard, cpu load…

  → https://github.com/ekatsah/fosdemx/blob/master/diagnostic.py
- Log all apps output with logging module

  → https://github.com/ekatsah/fosdemx/blob/master/settings.py#L16
- Don’t stream your log to the mothership (possible avalanche)

  → https://github.com/ekatsah/fosdemx/blob/master/_log.py#L14

Login & shell

- Why not python?
- Ask for password, check it, fork an interactive interpreter?
- Declare `/usr/bin/python3` as default shell, in `/etc/passwd`
- Declare `/usr/bin/python3` as valid shell, in `/etc/shells`

Login & shell

```
 $ ssh root@10.0.0.2

root@10.0.0.2's password:

Python 3.6.3 (default, May  2 2018, 11:18:10)

Type "help", "copyright", "credits" or "license" for more information.

>>> from sv import stats_process

>>> stats_process()

[{'alive': True, 'cmd': 'klogd -n', 'name': 'klogd', 'pid': 118,
  'runtime': 19.9, 'total_runtime': 19.9, 'restart': 0},
 {'alive': True, 'cmd': 'syslogd -n', 'name': 'syslogd', 'pid': 119,
  'runtime': 19.9, 'total_runtime': 19.9, 'restart': 0},

 ...
]
```

Halt/reboot

- Init should get `SIGTERM`/`SIGUSR1`
- Need some kind of teardown script:
  - Stop all apps, in a good order
  - Syscall sync fs
  - Syscall halt/reboot

Halt/reboot

```
logger.info("stop all systems")
sv.stop_all()

logger.info("sync filesystems")
libc = CDLL("libc.so.0", use_errno=True)
libc.sync()

if should_reboot:
    # just reboot
    libc.reboot(0x1234567)
else:
    # just halt
    libc.reboot(0xcdef0123)
```

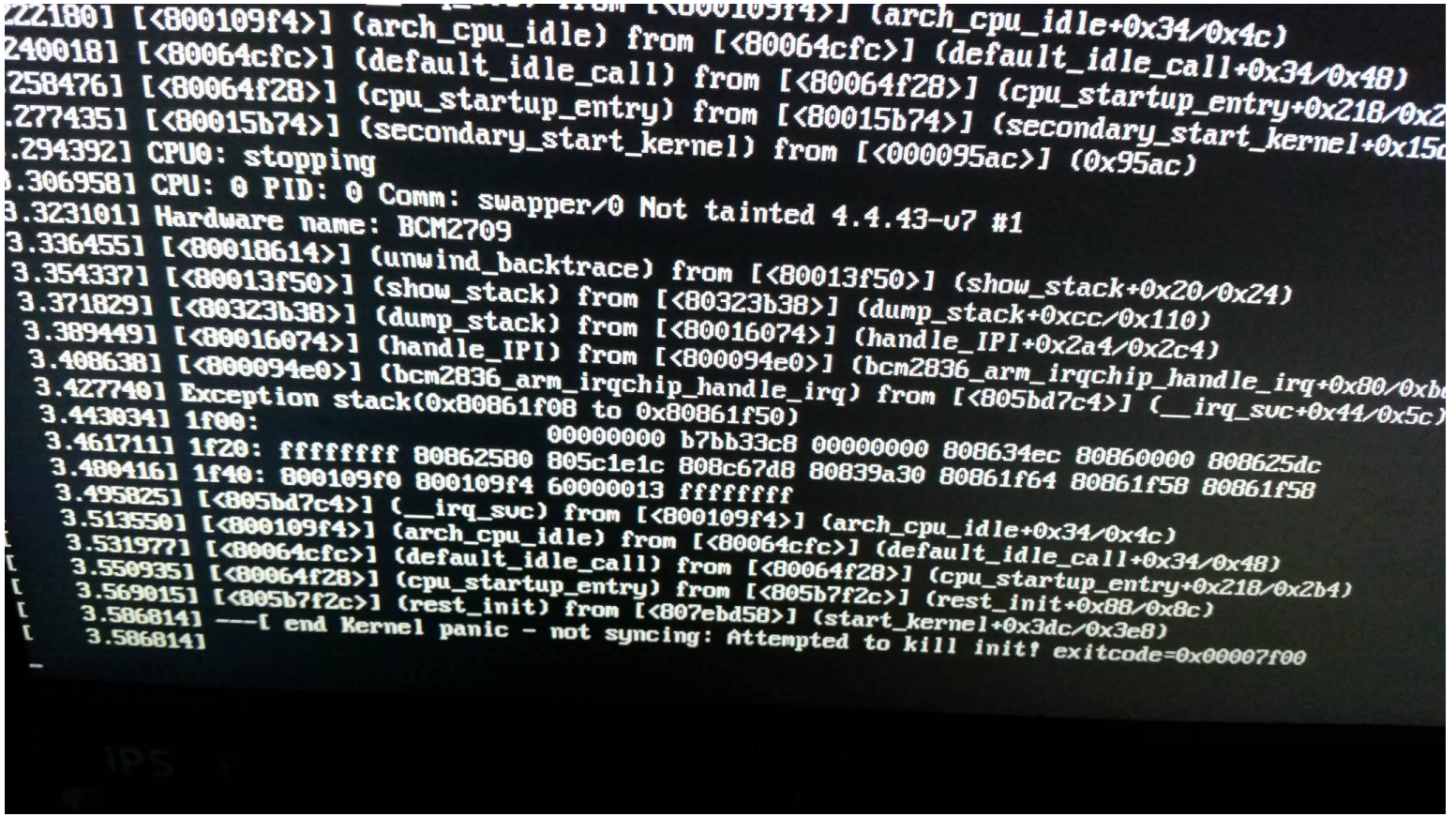
https://github.com/ekatsah/fosdemx/blob/master/stack.py#L190

Udev/Hotplug

- If you really need it, your hardware archi is probably wrong
- If you have “speculative” extension (eg: LTE + power saving), you can assume they will be always there
- The simpler way is to listen to uevent, grab the event relevant to you and act on it
- You can also use the hotplug helper in `/proc/sys/kernel/hotplug`

Testing

- S'il y a une erreur dans pid1 => kernel panic
- Anyway, error are annoying



Testing options

- Unit test to the rescue, up to a point
- Full integration testing: Qemu-system-arch, but you need to have a good description of your machine and its slow

Testing: lightweight integration

- Qemu-arch + chroot + unshare == love
  → https://ericchiang.github.io/post/containers-from-scratch/

```
# start binfmt daemon

echo ':arm:M::\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x02\x00\x28\x00:\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff\xff\xff
\xff\xff\xff\xfe\xff\xff\xff:/qemu-arm:' > /proc/sys/fs/binfmt_misc/register

unshare -f -p –mount-proc=<path>/proc/ chroot . ./qemu-arm /usr/bin/python3 /sbin/stack.py
```

- Mock everything:
  - Wifi: mac80211_hwsim
  - Serial input: pty
  - /sys: pyfuse

Last thoughts

- Be pragmatic: purity of vision is the biggest trap (a little bit of bash >> full python)
- Sometime you simply don’t need it, “if ain’t broke, don’t fix it”
- Strace is your best friend

NIH?

- This talk is not about reinventing the wheel, it’s about integrating existing components.
- Remove std component? We have here around ~1k loc of python: obviously we are leveraging A LOT of “invented elsewhere”

A little copying is better than a little dependency

– Rob Pike

In summary

- You should model your system with FSM
- You can write your own init/supervisor in python, the results are usually simpler and easier to debug for specific purpose system
- You should evaluate the possibility of using high level system to handle IPC instead of only posix primitives
- Container technology and emulation is great to test embedded system

Thank you!

IoT and leasure boats · Looking for a job? We are hiring!
Embedded · mobile · data → jeremie@sailsense.io