IoT edge cloud solution for embedded Linux devices

Contents

Todo list

1 Introduction

As this prject aims to implement an IoT like workflow, for deploying sandboxed applications on Linux embedded devices , we start by looking at existing solutions that might facilitate this kind of workflow. We

Afterwards we look at what extra features one these services provide which could improve the maintainability/longevity/flexibility of the workflow. Here we go through how to build an image, updating existing devices and provisioning a new device.

Afterwards we go through how to deploy a task to a device

Lastly we explore which attack vectors to secure against and how we do so.

1.1 Comparing existing IoT solutions

2 existing IoT solution for Linux embedded devices are considered. mender and pantacore.

Pantacore is written in C and makes use of LXC containers to deploy application, which is also written in C, giving the entire project an overall small footprint rootfs wise.
At the time of writing there are still several todos in their documentation. As exciting as it would be to contribute to this open source IoT solution, it is out of the scope of this project.

Mender is a lot more mature than pantacore and has the advantage that it can transform a debian image to a mender image with a fallback image, so this solution can be appended to any already existing workflow that makes use of debian images. This process as easy as it is, creates a rather large 8GB image.

Another consideration is that mender supports the yocto project which is a huge project that is used to create custom embedded Linux distributions. This seems perfect to facilitate the use of a patched kernel.

Based on the lacking maturity of pantacore and the fact that mender supports yocto, mender is chosen.

2 Building

A preliminary examination of using yocto versus using the mender-convert utility can be seen in table 1. For reference the raspbian buster lite image is approximately 1.8GB in size.
The table clearly shows that building an image using yocto with sstate caching is far superior in build time and image size, at the expense of a bit more setup time compared to the mender-convert utility. What is not shown here is that with yocto all source code for the build is downloaded so it requires more disk space from the build system.
Sstate caching is yocto genrating a cache of the compiled binaries, so as long no major architechtual changes take place there is no need to recompile a lot of binaries.

Table 1: Build comparison of the default mender raspberrypi yocto build and a mender-convert build of the default raspbian buster light image.

	Yocto with sstate caching	**Yocto**	**Mender-convert**
Build time	3m43.789s	83m11.750s	$\sim$ 20m
Image size	604MB	604MB	7.67GB

Using the yocto project also has the great advantage of having huge community behind it, with support for many different embedded boards which should make future hardware migrationse easier.
Committing to mender and the yocto project does increase the requirement to follow upstream yocto layers and update the yocto build to be compliant with upstream yocto.
To properly understand the advantages and disadvantages of using yocto, here is a short introduction.

2.1 Yocto

Yocto is build system for creating linux distributions for embedded devices. It relies on meta-layers each modifying the resulting image. These meta-layers can contain board specific settings, applications, modifications to other layers and so forth.

The primary files in a meta-layer are the .bb and .bbappend files.
These files follow a strict naming convention in the form of `{name}_{version}.{bb|bbappend}`. For an example see section .

skriv afsnit

2.1.1 Syntax

Variables can be initialised in 3 ways see listing 1.

```
# super weak initialiser
VAR ??= "1"
# weak initialiser
VAR  ?= "2"
# normal initialiser
VAR   = "3"
```

Listing 1: Initialisers in bb files.

Functions take the form `do_compile(){}` and a standard bb recipe contain the default functions `do_configure(){}`, `do_compile(){}` and `do_install(){}`.

A function can either be invoked inside a task or it can be promoted to a task and injected into the task chain.
Function can be written in shell or python.

2.1.2 Layers

The default layers for mender-raspberrypi

- meta-poky
- meta-yocto-bsp
- meta-oe
- meta-python
- meta-networking
- meta-multimedia
- meta-raspberrypi
- meta-mender-core
- meta-mender-raspberrypi
- meta-mender-demo

Though it should build without the mender demo layer.

2.1.3 Yocto layers

```
meta-mclur
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
└── recipes-kernel
    └── linux
        ├── linux-raspberrypi
        │   └── 0001-Canged-usbduxfast.patch
        └── linux-raspberrypi_%.bbappend
```

Each layer contains recipes e.g. .bb and .bbappend, these recipes contains instruction for building packages, which can the be included in the final image using the `IMAGE_INSTALL` variable.

2.1.4 Migrating to yocto

The MCLUR project uses a custom kernel versino of the usbduxfast kernel driver.

```
devtool modify linux-rasberrypi
```

```
<replace usbduxfast.c>
```

```
devtool finish -m patch linux-rasberrypi ../meta-mclur
```

This will generate a patch and a correctly named `.bbappend` file that will apply the patch to the rpi-kernel.

2.1.5 Workflow

```
project
├── build
│   └── conf
│       ├── bblayer.conf
│       └── local.conf
├── upstream-sources
│   ├── meta-layers
│   └── ⋮
└── meta-mclur
    ├── conf
    │   └── layer.conf
    └── recipes-kernel
        └── linux
            ├── linux-raspberrypi
            │   └── 0001-Canged-usbduxfast.patch
            └── linux-raspberrypi_%.bbappend
```

3 Deployment

3.1 Provisioning a new device

Assuming rpi3b+ as to always just run PXE boot

3.2 Updating existing device

- Bandwidth
- Devices downtime

3.3 Tasks

Tasks are deployed using the mender update module and deploying a task have been made idempotent to not insure that a device will not try to execute the same task twice over.

4 Security

The first attack vector that is considered is tampering with device, where an unsolicited 3rd party obtains physical access to the deployed device and transmit false data back.

The second attack vector that is considered is spoofing, where in an unsolicited 3rd party pretends to an already deployed device.

Lastly we consider a situation where an attacker have gained control over the internet connection and is trying to incorporate the deployed device into a botnet.

4.1 Tamper protection

4.2 Sign returned data with solokey

Verify hardware has not been replaced

4.3 Certificates and keys

Mender

Can be created from default debian images, so its less intrusive to existing workflow Images as a result of this is huge (requires 8 GB) https://docs.mender.io/2.2/getting-started/quickstart-with-raspberry-pi support Yocto as part of their layers, which should reduce image footprint Will require a new workflow, toolchain, etc. Huge community in both Yocto and Mender

Pantahub

Written in C and make use of LXC containers which are also written in C giving the complete project an overall very small footprint Less mature, at time of writing there still several todo in their documantation. Also a smaller community.

Yocto

Uses layers, makes building images for specific usecases easier as necessary layers are included. Greater set of requirements, for host build system, Arch linux not working. Allows for delta updates in mender to reduce bandwith in Over-The-Air updates.
Makes it easier to transistion hardware.
Cross-Prelink: - Generate link tables for dynamic linking.

Their crosscompiler is running on docker. Had make own docker image for poky to compile. Recommended setup is also to build on host. Look into qemu building.
Bitbake scales very well with multiple cores. Poky base image takes up alot of space 50GB plus to build from scratch.

Guy describes using yocto for RPI
https://jumpnowtek.com/rpi/Raspberry-Pi-Systems-with-Yocto.html

SSTATE CACHING, working - Must be turned on in `conf/local.conf`.

Tests

- Build time
  Yocto vs Mender convert
- Image size
  Yocto vs Mender convert

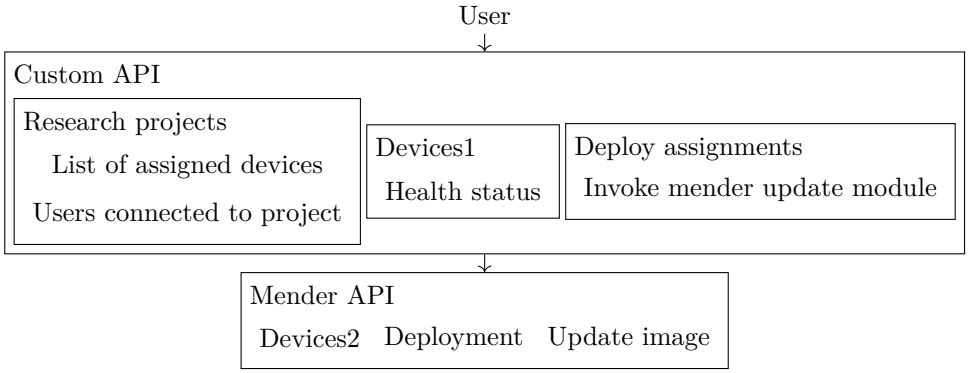

Figure 1: A caption