IoT edge cloud solution for embedded Linux devices

Author:
Aske Bækdal Møller

Supervisor:
Tórur Andreassen

Email:
asmoe16@student.sdu.dk

University:
University of Southern Denmark

Abstract

This project aims to determine the feasibility of implementing an IoT like workflow for deploying sandboxed applications to the MCLURS system developed by SDU.
This is done by looking at existing IoT solutions mender and pantahub and attempting to migrate the MCLURS project.
Mender is a viable IoT solution for Linux embedded devices, with create emphasis on robustness and security.
As some libraries could not be made functioning under the OpenEmbedded layer scheme there is a time penalty to migrating to yocto, but as mender supports converting debian images there is a opportunity for a multi step migration process.

Contents

Todo list

1 Introduction

As this project aims to implement an IoT like workflow, for deploying sandboxed applications on Linux embedded devices. In this project only raspberrypis are used, but with some tooling it should be made to work on many other devices.

There already exists open-source solutions which can be taken advantage of, as it is generally a bad idea to write security yourself.
To make the project more goal oriented it aims to port the MCLURS project to this workflow. This is done both to prove the workflow works, but also to delve deep into the workflow to discover any quirks that might mike this workflow unsuitable.
The report is divided into 3 main sections, building, deployment and security.
Building covers everything up until using the system to deploy device and tasks. Deployment describes how device are provisioned, deployed and given tasks. Security covers which security concerns have been taken into account.
To further specify the project a series of requirements are listed in section 1.2.

1.1 Existing solution

The existing solution has no network connection and needs to be physically accessed to be updated or to retrieve data from it.
The OS is a debian variant with a custom kernel.

1.2 Requirements

- It should be possible to deploy an application to a specific PI in the field.
- It should be possible to re-deploy to a specific PI in the field.
- It should be possible to deploy an application and run the PI in the field, offline
- It should be possible to get the log from the PI while it's in the field (and is online)
- There should be a built-in mechanism that handles software failure (rolls back to the last working version)
- The solutions must be able to handle a custom kernel (as it needs to support the custom kernel module made by John)
- The applications must be able to talk with the hardware (I2C, DAC etc.)
- Deploying the initial image/stub/whatever to a PI must be fast, easy and as automatic as possible (PXE boot, etc.)
- Registering/discovering the new PI must be fast, easy and possible to automate.
- The system must handle different namespaces/accounts/groups in order to be able to handle multiple projects with different project owners.
- The solution must not rely on being able to SSH/contact the PI directly - all communication with the PI must happen through a web interface.
- All configuration for a certain application(s) must not rely on custom config of the stub/host-image - all configuration should be included in the container/image. (kernel modules are an exception to this)
- Persistent data (sound recordings) should be saved on an external harddrive (outside the container)
- Security must be part of the solutions design, so the PIs can't be used as botnet, mining, etc.

1.3 Comparing existing IoT solutions

2 existing IoT solution for Linux embedded devices are considered. mender[4] and pantacor[**?**].

Both solutions support fallback in case an update fails. They both let the deployed device poll for updates as to not have open ports on the device.

Pantacor 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 pantacor 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.

1.3.1 Yocto

Migrating to the yocto project also brings other less obvious advantages. If the mender project unexpectedly should cease to exist another solution could be implemented in a new layer. Yocto is also a very stable project with both Cisco and Microsoft Azure using it it.

Based on the lacking maturity of pantacor 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 bitbake files with the extension .bb and .bbappend files.
These files follow a strict naming convention in the form of `{name}_{version}.{bb|bbappend}`.

2.1.1 Software setup

The setup used is the crops/poky docker container as the current build host OS is not suitable for building yocto images and using docker image saves the headache it can be to setup a cross-compilation toolchain correctly. It would be more preferable to have a native build host, as that can make bitbake recipes that makes use of libraries of the build host easier to manage.

Table 2: Yocto termonology

Term	Description	Naming convention
Layer	A set of recipes, classes and a layer config file.	`meta-<LayerName>/`
Recipe	An instruction on how to build one 1 more packages. Contains source URL, checksums, build time dependencies and runtime dependencies.	`<RecipeName>.bb[append]`
Package	Application generated by recipe. Can be only needed during build time.	

At the time of the project start mender 2.2 was the latest stable release and therefore it is the one being used. The latest yocto version that mender supports is 2.7.3, codename warrior. The latest yocto release is 3.1(dunfell), but looking through the OpenEmbedded layer index[6] it can be seen that a lot of layers have yet to be made compliant with yocto 3.X, so it does not seem to be a sign of staleness on menders part.
A git repository has been made with the correct version of all layers needed for this project and can be found on github[2].

2.1.2 Migrating to yocto

The primary applications of the MCLURS project is publicly available on gitlab[3]. To start new layer, the configuration files could be written by hand, but OpenEmbedded har at tool `bitbake-layer` which can create a template for a new layer , using the command `bitbake-layer create-layer <path>` . From here a best guess recipe can be made using `devtool add <git-url>` . This creates a workspace workspace folder like this:

```
workspace
├── recipes
├── sources
└── ...
```

`devtool` will automatically attempt to create a recipe that builds the project. The mclurs git however does not have a `Makefile` in the root of the repository. Instead the make files are located in `mclurs-1.0/` and `libmclurs-snap-perl-1.0/` . The most correct thing to do here is to have separate recipes for each sub-project, ideally dividing them into separate git repositories. As to not disturb any current workflow an extra `Makefile` is provided, that builds and installs the sub-projects. This also allow the use of the `autotools.bbclass` which provides default pre-compile and compile functions. All that is left to do is to tell what package the generated file belong. The full bitbake recipe can be seen in listing 1.
Normally when creating a perl recipe it usally enough to inherit from the `cpan.bbclass` to setup the correct build environment, but as `libmclurs-snap-perl-1/` only has an install target and no compiling is not needed, using a normal `Makefile` suffices here.

There is also a custom version of the usbduxfast kernel module which the mclurs system make use of. It resides in `mclurs-1.0/adc/kernel/` . Usually the `usbduxfast.h` file is replaced in the linux kernel is replaced with this custom one written by John Hallam. There 2 ways to go about this; either create a patch or prepend an action to the pre-compile function that replaces the usbduxfast module. Both solutions are viable, but the most commonly accepted solution is to make a patch. An easy way to do this is again to use `devtool` . `devtool modify linux-raspberrypi` will populate `<WORKDIR>/build/workspace/sources/linux-raspberrypi/` with the source from the linux-kernel used for the raspberrypi. Then its just replacing the usbduxfast file, git committing the change and using `devtool finish -m patch linux-raspberrypi <WORKDIR>/meta-mclurs` . This generate a `.bbappend` file in the meta-mclurs layer and a patch file that will then be applied.

The mclurs project is runtime dependent on zsh, daemontools, runit and some perl modules. There is already recipe for both zsh and daemontools, but not for runit. The trouble with runit is that is an init system would replace systemd that mender is dependent on. The best solution would be to either port mender to runit or mclurs to systemd. As a middle of the road alternative a runit recipe that runs as a systemd service is created using the same workflow as the mclurs project. The recipe is based on an AUR makepkg file [1].

The `libmclurs-snap-perl-1.0` has a perl dependency tree which can be seen in figure 1

```
LICENSE = "Unknown"
LIC_FILES_CHKSUM = "file://libmclurs-snap-perl-
↪  1.0/debian/copyright;md5=3d7a1388251a016265b06f7fe77d9699
↪  \
                    file://libmclurs-snap-perl-1.0/debian/libmclurs-snap-
↪  perl/usr/share/doc/libmclurs-snap-
↪  perl/copyright;md5=3d7a1388251a016265b06f7fe77d9699
↪  \

↪  file://mclurs-1.0/debian/copyright;md5=4000a2a9c915abe613cf1ee84d51d60f"

SRC_URI = "git://gitlab.com/esrl/mclurs.git;protocol=https \
    file://Makefile;subdir=git"

# Modify these as desired
PV = "1.0+git${SRCPV}"
SRCREV = "ff0f2ab3bac9147bb54895469f9a0c0965432923"

S = "${WORKDIR}/git"

inherit module autotools-brokensep

RDEPENDS_${PN} = "zmq-perl daemontools runit-systemd"

FILES_${PN} = "${bindir}/* ${sbindir}/* ${datadir}/*"
```

Listing 1: MCLURS bitbake recipe.

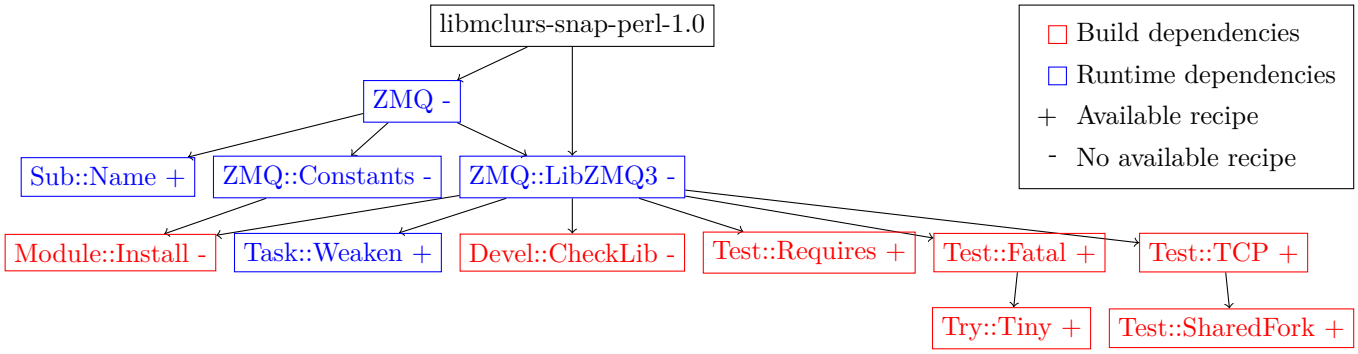

Figure 1: Perl dependency tree for libmclurs-snap-perl-1.0.

The dependencies are gathered using a tool from cpan[7].
Unfortunately it was not possible to port ZMQ::LibZMQ3[8] to yocto mostly due to lack of understanding of perl build system.

The resulting MCLURS layer is availe on github[5]. The packages provides by the MCLURS layer are listed her:

- devel-checklib-perl
- file-remove-perl
- mclurs (not fully functional in runtime)
- module-corelist-perl
- module-install-perl
- module-scandeps-perl
- runit-systemd
- yaml-tiny-perl

- zeromq
- zmq-constants-perl
- zmq-libzmq3-perl (not building)
- zmq-perl (not building)

2.1.3 Easier migration option

Instead of creating all these recipes mender has an update module that supports docker passing docker containers. The value here that not as much dependency resolving is necessary.
Docker would fill all the requirements regarding sandboxing hardware access and so on, but the cost would be a greater overhead while running the applications, which in turn means a greater power consumption.
If the application that runs on the deployed device changes frequently a container solution would make a lot of sense, but as this is not the case with the mclurs system this solution is suboptimal, but easier from a migration stand point. An example of a MCLURS docker container can be seen en section 3.4

3 Deployment

3.1 Provisioning a new device

Assuming rpi3b+ as to always just run PXE boot. Though if using an older rpi unit use `bootbin.sh` to setup an sdcard up with the bootcode.bin file which enables older rpis to PXE boot.
PXE boot server is hosted on an rpi. A new device will boot a default raspbian buster image which mounts an nfs share from the development host which have the current sdcard image that should be deployed. On the same share there is a script, `boot.sh` , which is run using a cron job during startup. This is illustrated in figure 2.

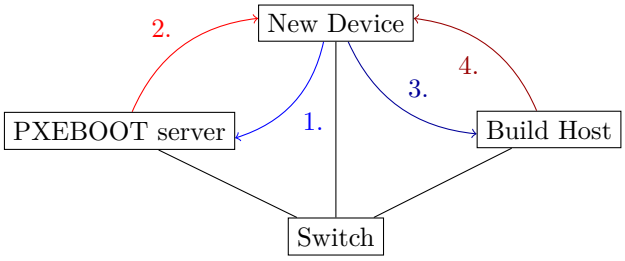

Figure 2: 1. PXEBOOT; 2. Boot default RPI image; 3. Mount NFS share; 4. Run `boot.sh` script

When a device boots the mender image for the first time it will show up as pending on the mender host interface as can be seen in figure 3. Here it can be accepted as a authorized device. This is under the assumption that local network on which the developer host attached is secure.

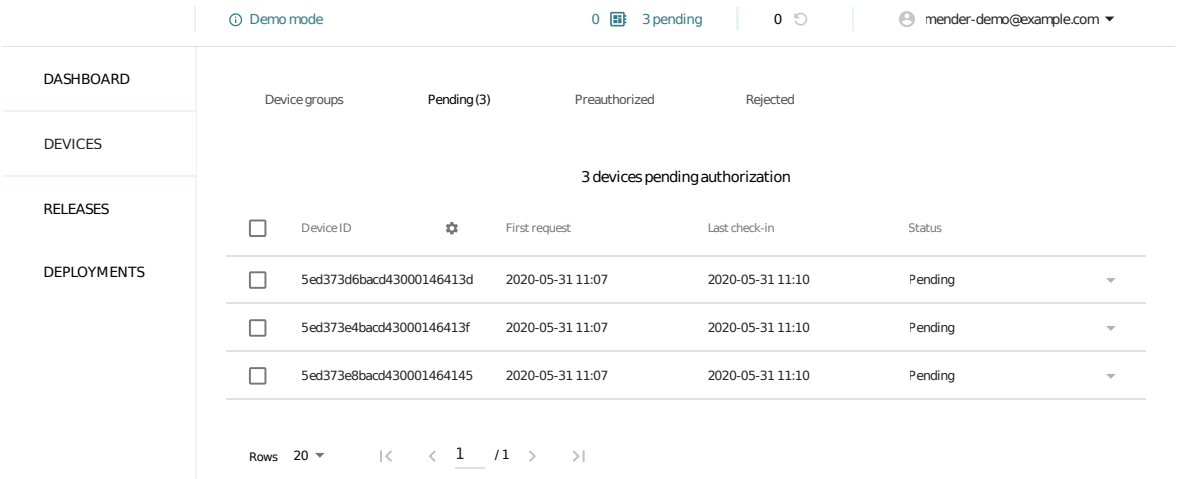

Figure 3: Mender interface when device is pending

3.2 Updating existing device

Updating a device is taken of by the mender server. When compiling a sdcard image using yocto, a mender artefact is also generated, this can be uploaded to the mender server. The artefact can then be assigned devices or groups, where any device in an assigned group will receive the update the next time the deployed device polls the mender server.

To ensure robustness mender makes use of what they call an A/B rootfs. Of rootfs A is version 1.0 adn B is 2.0, then B is in use if it has been committed. Then if the device updated A will be overridden to the new version and the deployed device will try to commit the new rootfs.
An overview of the 9 possible states a m,ender client can be seen in figure 4. A mender artifact can both be a update for a mender update module or a rootfs update. With this system mender can deliver robust and reliable updates for deployed devices.

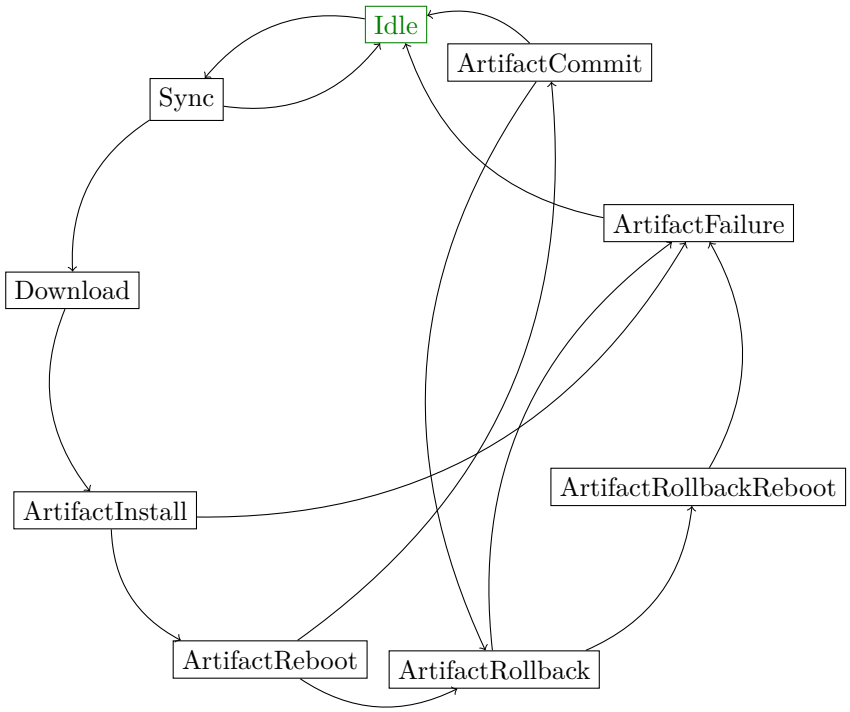

Figure 4: Mender states during update

This system with states also allow for great customizability when creating update modules.

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.
A mender update module is an executable located in `/usr/share/mender/modules/v3` which takes 2 arguments, state and files.
For this project we assume that any needed program is existing on the deployed device, so a update module is made that adds tasks as cron jobs.
Tasks are visible in the mender inventory.

3.4 Docker

Instead of porting projects mender already has a docker update module. This of course reduce the requirement for porting project at the expense of greater overhead, connected with running containers. The custom kernel module is still needed on the deployed device, but other than that it only needs the docker service.
A mclurs docker file can be seen in listing 2.
The docker container can then be build and pushed to dockerhub. A mender artifact can then be generated as shown in listing 3. The artefact can be uploaded the mender server and from there a deployment can be created to any device matching `${DEVICE_TYPE}` .
Demo docker with c-sense-hat interacting with hardware.

```
FROM perl:slim-buster
CMD ["bash"]

RUN apt-get -q update \
        && apt-get -qy install perl daemontools libzmq3-dev gcc
RUN apt-get -qy install runit

RUN yes yes | perl -MCPAN -e 'ZMQ::LibZMQ3'
RUN perl -MCPAN -e 'ZMQ'

RUN apt-get -qy install libargtable2-dev libcomedi-dev libcap-dev

COPY mclurs /mclurs
WORKDIR /mclurs/mclurs-1.0
RUN make
RUN make install

WORKDIR /
RUN rm -rf /mclurs

RUN apt-get -qy remove --purge gcc \
        && apt-get -qy autoremove

CMD ["bash"]
```

Listing 2: A docker file that enables the MCLURS project to run in a dockerdocker container.

```
docker build . -t mclurs
docker push ${ID}/${REPO}:mclurs
docker-artifact-gen -n ${ARTIFACT_NAME} -t ${DEVICE_TYPE} -o ${OUTPUT_PATH}
↪  ${DOCKER_IMAGES}
```

Listing 3: Generate mender artefact for the docker mender update module.

4 Security

As soon as a device has an active internet connection security is a concern. The primary concern is the device being hijacked and used in a botnet for malicious purposes.
For the MCLURS project it is also a concern to verify a device such that it is not possible for anyone to deploy a device and report research data. This also includes that it should not be possible to tamper with a device, at least not undetected.
Most security regarding the deployed devices are handled either by upstream as it is a bad idea to do security yourself. The only concern not covered by upstream is tamper protection/detection and theft.
Theft is not covered here.

4.1 Tamper protection

Assuming the deployed device is in a box, the box can be wired in such a ways that any opening of box will trigger an event on the device.
This will not prevent tampering, but will allow server side detection of tampering.
To demo this effect a small C program is written that utilises the button on the sense hat module.
When the event i triggered a time stamp overwrites the current time stamp. Time stamp is then available in the mender inventory. When this message then changes in the mender inventory it can then be verified whether access to the deployed device was legitimate or not.
An example returned from the mender API can be seen in listing 4.

```
{
  "id": "5ed373e8bacd430001464145",
  "attributes": [
    {
      "name": "rootfs_type",
      "value": "ext4"
    }, {
      "name": "mac_eth0",
      "value": "b8:27:eb:c5:de:49"
    }, {
      "name": "ipv4_eth0",
      "value": "192.168.0.41/24"
    }, {
      "name": "device_type",
      "value": "raspberrypi3"
    }, {
      "name": "tamper_date",
      "value": "2020-05-31T14:42:09.02Z"
    }, {
      "name": "mender_client_version",
      "value": "2.2.0"
    }
  ],
  "updated_ts": "2020-05-31T14:42:09.02Z"
}
```


Listing 4: Mender device inventory example in JSON format.

4.2 Over-The-Air updates

Mender handles OTA via device polling, as in the device polls the mender server for updates. This ensures that no port on the deployed device needs to be open, greatly lowering the risk of a deployed device being hijacked. The device verifies the mender server using SSL certificates. This should overcome any man in the middle attack.

5 Discussion

The solution found here using mender and the yocto project to provide a feasible IoT edge cloud solution does indeed work. Though it has a greater footprint then complete custom solution but that is to be expected for a more generic solution. Using a large open source project also benefits from more auditing from its users compared to a smaller custom solution. It should be expected that a larger project should be more stable and secure. Of course one could argue for security by obscurity, but that is generally frowned upon and would only work as long as the project is very small.
The current project is not as easy to maintain, as the previous distribution method, mainly because the previous solution was not connected to the internet. The chosen solution does incur some more maintenance compared converting a debian image. All the benefits of using the yocto project are not apparent in this project. Going forward it would be easier for MCLURS system to branch out to use other devices and even use crossover SoCs with multiple hardware architectures to get more specialised sound recordings.
Migrating to the yocto project is not penalty free, as it requires some more specialised knowledge about how yocto does things and the multitude of ways a single thing can be achieved. To help accomadte migration the yocto project has their `devtool`, which can automate most of the migration and create the initial recipes. It doesn't do everything and knowledge about how the yocto project operates is still needed.
As seen in this project close familiarity with the software being ported to yocto is a huge benefit as dealing with new workflows and new build systems can make for a confusing debugging experience.
An advantage that choosing the mender project is that they offer to host the mender server, which can reduce needed maintenance hours in maintaining server security.

6 Conclusion

An IoT edge cloud solution for Linux embedded device is presented in this project. It is should be feasible for any embedded software developer to migrate a project to the yocto project. The solution does not incur the smallest maintenance penalty, but it does move the critical maintenance points to a large open-source community.

It also eases any future expansion to other embedded devices, conserving the same workflow across multiple hardware platforms.

References

[1] Aur: runit-systemd. Link.

[2] Git with all needed layers to build this project. Link.

[3] Mclurs project git. Link.

[4] Mender homepage. Link.

[5] The meta-mclurs git. Link.

[6] Openembedded layer index. Link.

[7] Perl dependency tool. Link.

[8] Perl module: Zmq::libzmq3. Link.