GSOC'24 Proposal

Building support for an A/B partition scheme-based update for Uptane Client

Contact Information

Name : Shivam Singhal

Email : singhals.3316@gmail.com

Github : singhals163

LinkedIn : singhals163

Phone : +91-9368722259

Address : 429/7 Nehru Nagar, Garh Road, Meerut, U.P. 250002

Time Zone : Kolkata, India(GMT+5:30)

University : Indian Institute of Technology Kanpur

Degree : Bachelor of Technology, Electrical Engineering (Fourth year)

Commitment : 35-40 hrs/week

Other Commitments: Remote Summer Research Project

Resume : Link

Project Information

Title	:	Building support for an A/B partition scheme-based update for Uptane Client
Time	:	350 hrs (12 weeks)
Difficulty Level	:	Difficult
Required Skills	:	Systems-level programming in C++

Abstract

Aktualizr is an Uptane client written in C++, targeting embedded Linux systems. Uptane’s core functionality is securing and validating software updates in very security-sensitive and safety-critical systems. Aktualizr combines two important areas of functionality: implementing Uptane to actually validate software updates and then installing those software updates. However, the mechanics of installing software updates on embedded Linux systems are usually quite complex, so Aktualizr hands software artefacts off to other installers/libraries after they have passed all Uptane security checks. To do this with maximum efficiency, it's usually important to integrate quite deeply with the system responsible for managing the system updates.

Currently, aktualizr only supports OSTree as a method of installing Linux OS updates (including kernel, initramfs, device tree, rootfs). There are other embedded Linux OS updaters, such as SWUpdate and RAUC, that are based on an A/B partition update scheme, a popular choice for embedded devices and automobiles. However, these projects do not yet support Uptane verification of their software updates.

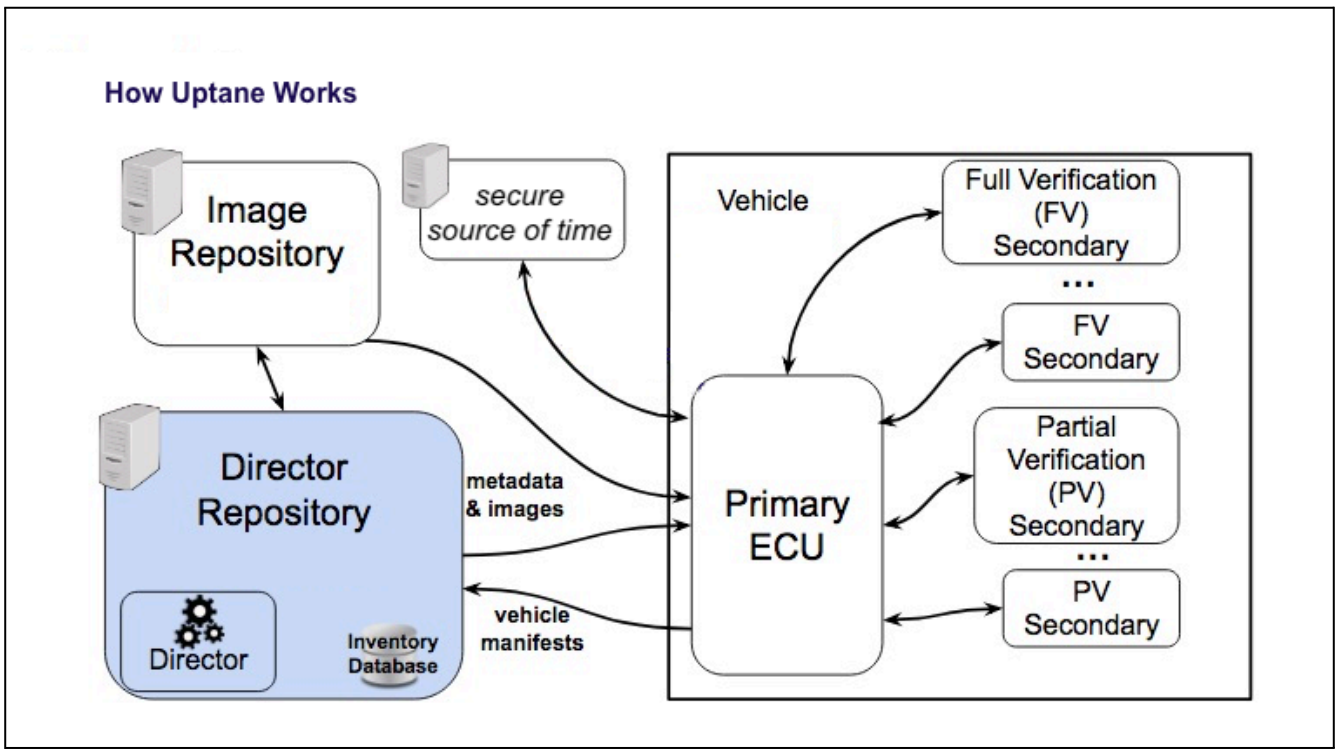
This GSoC project would contribute to Aktualizr's implementation of support for an A/B partition-based update method, preferably integrating with SWUpdate or RAUC. This would enable the manufacturers who use A/B based updates to be able to utilise the security features offered by Uptane and thus ensuring a wider user support for Uptane.

Introduction

Uptane[1] [3]

Uptane is the first software update security system for the automotive industry capable of resisting even attacks by nation-state level actors. It is designed so that the security of software updates does not degrade all at once, but follows a hierarchy in which different levels of access to vehicles or the automaker's infrastructure must be gained before irreparable damage can be inflicted.

Uptane is a standard set of rules to follow that will make your software update system resilient to a wide variety of real-world, long-term attacks.



Uptane Architecture [3]

The above image from the Uptane Standard for Design and Implementation gives a brief overview of Uptane's architecture for ensuring safe updates. Uptane utilizes multiple servers, known as repositories, to

[1] Some parts of this section have been taken from the Uptane's official website: Uptane Documentation

provide security through the validation of images before downloading. This diagram illustrates how the checks and balances of this system work. The connected components on the right hand side of the diagram are on the vehicle, while the components on the left hand-side represent the repositories. As our project is primarily concerned with the client side of the model, I'm going to briefly talk about that in the following section.

Uptane Client

Uptane Client is a program that runs on an embedded device to ensure the following tasks:

- Communicating with the Uptane Server to get notified of new updates
- Sending information about the current image installed on the embedded device so the Server can compare and identify any new update
- Verifying the new images to be installed on the embedded device
- Preparing the embedded device to run the updated version of code after a reboot
- Verifying whether the update was successful and updating the new root to be the default root of the filesystem, thus enforcing the update
- Initiating a rollback mechanism in case of faulty updates

Literature Review

There are multiple methods to implement updates to the kernel image of a system. Of them, the abstract talks about two major approaches:

1. OSTree: The original approach, follows a git based update strategy
2. SWUpdate and RAUC: works on A/B partition-based updates

Each of the approaches has its own benefits; it depends on the developer of the underlying embedded system which method they choose to facilitate the update process. Currently, the Uptane client (Aktualizr) does not implement A/B partition-based updates, and thus, the embedded devices employing this methodology can't use the additional security features provided by the Uptane project.

A/B partition-based updates[2] [1]

In A/B partition-based update mechanisms, updates are incrementally configured in separate partitions. If partition A is currently active, updates are installed in partition B, which will then become the active partition. In case there is a failure or a faulty update, we can always revert back to partition A. Moreover, A/B system updates provide the following benefits:

- OTA updates can occur while the system is running. The only downtime during an update is when the device reboots into the updated disk partition.
- After an update, rebooting takes no longer than a regular reboot.
- The user will not be affected if an OTA fails to apply (for example, because of a bad flash). The old OS will continue to run, and the client is free to try the update again.
- If an OTA update is applied but fails to boot, the device will reboot back into the old partition and remain usable while, the client decides whether to run t the update again.
- Any errors (such as I/O errors) affect only the unused partition set and can be retried. Such errors also become less likely because the I/O load is deliberately low to avoid degrading the user experience.
- Updates can be streamed to A/B devices, removing the need to save the package/image on a storage device before installing it. Streaming means it's not necessary for the user to have enough free space to store the update package on /data or /cache.
- dm-verity guarantees a device will boot an uncorrupted image. If a device doesn't boot due to a bad OTA or dm-verity issue, the device can reboot into an old image.

Background Research

Aktualizr has a standard framework for implementing the Uptane client using any packagemanager. The following section explains in details the key requirements for implementing a new client and how SWUpdate or RAUC will be able to fit in this definition.

---

[2] A/B (seamless) system updates | Android Open Source Project

The Uptane Client

Aktualizr currently employs `OSTree` as its software update system after running the security verification checks on the Target image.

To get started with the project, my first task was to review the client's Interface. The general implementation of the Aktualizr client given by *PackageManagerInterface* can be extended as required to implement the new update mechanism for the embedded systems. The interface has the following main methods that need to be implemented:

1. **getInstalledPackages()** : Returns a list of packages installed on a device and reports it to the back-end via an HTTP API
2. **getCurrent()** : returns a Target object containing the details of the currently installed image
3. **fetchTarget()** : fetches the new image file to be installed on the system
4. **install()** : installs the new package
5. **completeInstall()** : reboots the system in order to run the new update installed
6. **finaliseInstall()** : checks if the install was successful and if the installed update's hash matches the expected hash value

Apart from these steps, several helper functions and different methods are part of the PackageManagerInterface and need to be implemented as part of the new Client that we aim to develop through this project.

One major objective here is to implement the SWUpdate or RAUC Client using the given framework only (for reasons of **backwards compatibility**) and not creating any new methods in the abstract class *PackageManagerInterface*, as these methods will have to be implemented for other Clients as well, which is not a desired solution.

SWUpdate[3] [5]

The next task was to explore the features and properties of SWUpdate and find the correct ways to integrate it with the current framework of Aktualizr. The documentation of SWUpdate is pretty detailed and

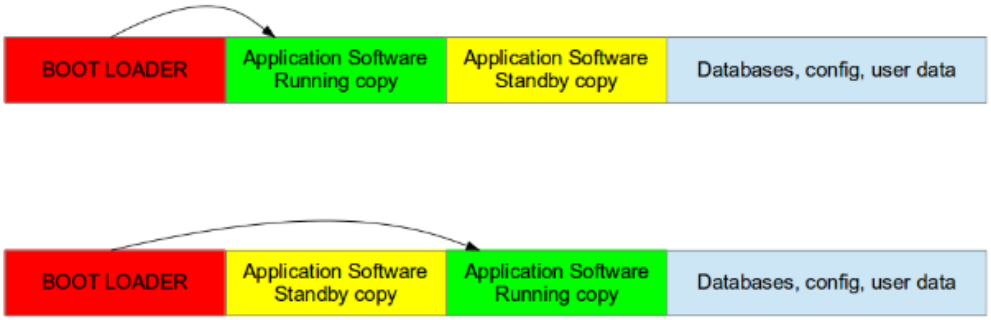
[3] SWUpdate Documentation

provides answers to preliminary questions as to where to start, what features to look for, and what to ignore for an initial implementation of this project.

The following key features/ideas form the basis of integrating SWUpdate with Aktualizr[4]

1. Double copy with fall-back

- Each copy must contain the kernel, the root file system, and the next component to be updated. It requires a mechanism to identify which version is running.
- SWUpdate should be inserted in the application software, and then the application software will trigger it when an update is required. The duty of SWUpdate is to update the *standby copy*, leaving the *running copy* of the software untouched.



2. sw-description

- This is a generic XML file used to define the properties of the new SWUpdate image scheduled to be installed on the embedded system
- We can write our own parser for sw-description if needed
- It will contain all the information as to which partition the new image should be installed to and the hash of this new image
- We can parse this file to extract only the information relevant to the project's scope now and later build other features around the basic framework as and when, the need arises

3. File-System Handlers

- SWUpdate needs handlers for installing images on different FileSystems

[4] Some parts of the key points about these features have been taken from the SWUpdate documentation

- There is an exhaustive list of already implemented handlers for mainstream filesystems. These include:
  - Flash devices in raw mode (both NOR and NAND)
    1. UBI volumes
    2. UBI volumes partitioner
    3. raw flashes handler (NAND, NOR, SPI-NOR, CFI interface)
    4. disk partitioner
    5. raw devices, such as an SD Card partition
    6. bootloader (U-Boot, GRUB, EFI Boot Guard) environment
    7. Lua scripts handler
    8. shell scripts handler
    9. rdiff handler
    10. readback handler
    11. archive (zo, tarballs) handler
    12. remote handler
    13. microcontroller update handler
- We can write our own custom handler for any other specific filesystem that we may need

4. Installed-directly flag

- SWUpdate uses the flag *installed-directly* to set whether to install the image directly to the new partition or make a temporary copy first. This is part of sw-description
- SWUpdate can be enabled for *zero-copy* (or streaming mode). in which the incoming SWU is analyzed on the fly and installed by the associated handler without any temporary copy. If this is not set, SWUpdate creates a temporary copy in *$TMPDIR* before passing it to the handlers

5. Pre-post install scripts

- These are scripts that the manufacturer may want to run on the embedded device before and after the update process
- As discussed, these may cause the signature of the image to change and thus may lead to some problems in Uptane verification of the installed image
- For this reason, after a discussion with the project mentors, the idea of running the pre- and post-update scripts has been kept out of

the project's current scope.

6. Write about verified images and hashes

- The images can be signed and have their own hashes which are part of the metadata contained in sw-description
- This information can be used to verify the image during the verification step of Uptane

7. Delta-updates

- Refer here

RAUC[5] [7]

RAUC has a framework similar to SWUpdate and was also explored during the pre-selection period. Deciding which one of them to implement in the final project is a crucial choice and will be made in the initial week of the project based on better user support and ease of implementation. Going a step further, we can also provide support for both the frameworks, as once the client is implemented for one of them, it is easy to tailor that client for the other framework. This can be an extended open-source project to continue the work later.

The following key features form the basis for integrating RAUC with Aktualizr[6]:

1. Update Bundles

A RAUC bundle consists of:

- File system image(s) or archive(s) to be installed on the system
- A *manifest* that lists the images to install and contains options and meta-information
- Possible scripts to run before, during or after installation

2. SquashFS image and direct streaming

[5] RAUC Documentation

[6] Some parts of the key points about these features have been taken from the RAUC documentation

- To pack the contents of RAUC Bundle together, these contents are collected into a *SquashFS image*
- This provides good compression while allowing users to mount the bundle without having to unpack it on the target system
- This way,no additional intermediate storage is required

3. HTTP Streaming

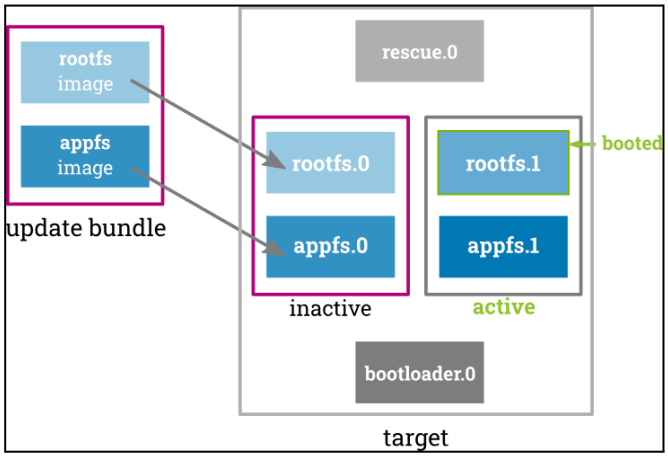
- Bundles can be installed directly from a HTTP(S) server, without having to download and store the bundle locally
- This is a key feature that is necessary for OTA updates

4. Slots

- Everything that can be updated is a *slot*. Thus, a slot can be a full device, partition, volume, or file.
- To let RAUC know which slots on the board must be handled, the slots must be configured in a *system configuration file*.
- This file is the central instance that tells RAUC how to handle the board, which bootloader to use, which custom scripts to execute, etc.
- This includes the slot description names, for example, which file path allows access to the slot, what type of storage or filesystem to use, how it is identified from the bootloader, etc.

5. Target Slot Selection

- We have to form a correct mapping of images contained in RAUC bundles to inactive slots, or else we may end up updating a running slot
- RAUC defines different *slot classes*. A class describes multiple redundant slots of the same type. For example, one class could be for root file system slots, while another class might be for application slots.
  Note: RAUC allows multiple slots per class, unlikea classic A/B partition-based system
- Multiple slots of different classes can be grouped as a *slot group*. *For example*, a system may have two redundant rootfs slots and two redundant



application slots. Then they can be grouped together to compose a fixed set of rootfs and an application slot.
- RAUC detects the active slots by using mount information of partitions, and thus streams the image to an inactive slot.

6. Slot Skipping
- We can skip a slot by matching the checksum of the currently installed image in that slot with the new image to be installed
- If the checksum is the same, the incoming image is not installed in the slot
- *This is necessary in case only a few of the slots needs to be updated in the slot group*

7. Boot Slot Selection - Configuring Bootloader
- RAUC cannot select the slot to boot itself, but provides a well-defined interface and abstraction for interacting with different bootloaders (e.g. GRUB, Barebox, U-Boot) or boot selection methods
- In order to allow RAUC to switch to the correct slot, its system configuration must specify the name of the respective slot from the bootloader's perspective
- You also have to set up an appropriate boot selection logic in the bootloader itself by scripting (as for GRUB, U-Boot)

8. Update Handlers for various FileSystems
- RAUC needs to know how to write an image to a slot
- This is governed by the corresponding filesytem of the slot
- A handler has to be provided for each fs for RAUC to utilize while writing the update
- *Update Handlers* solve this problem in RAUC
- RAUC finds the appropriate Update Handler by matching the slot type and image type
    - Slot type is defined in RAUC system configuration file
    - Image type can be found by checking the extension of the image

9. Boot Confirmation and Fallback mechanism
- As detecting an invalid boot is difficult, we need to keep a hardware watchdog to detect if a boot was valid

- After update, if the boot is invalid, the hardware watchdog will trigger the code to change the configuration of the bootloader to load into the previous partition
- We can thus ensure a fallback mechanism in case of faulty updates

Objectives

1. Decide whether to use RAUC or SWUpdate as the package manager to implement the Uptane client
2. Design an Uptane Client based on Aktualizr's framework, which uses SWUpdate or RAUC to install the Target image
3. Ensure backwards compatibility of the solution
4. Implement the solution for a base case where we have a bootloader partition and two root partitions, namely the A and B partitions, needed to facilitate the update
5. Embed the information provided in *sw-description(SWUpdate)* or *manifest(RAUC)* in the current framework of Uptane::Target
6. Utilise functionality provided by the *direct-streaming* method to install the image directly to the Standby copy, ensuring space optimisation
7. Implement *Delta Updates* to download only the diff image from the currently installed image, making the solution more robust, space and time-optimised (extended feature)

Approach

I will divide the problem into 3 major parts:

1. Verifying the image to be downloaded
2. Efficiently installing the image and making further space and time-optimisations
3. Configuring the bootloader to boot into the new partition on the next restart and verifying the installed image

I. Verifying the image to be downloaded

- Design an approach for verifying hashes and reporting results under SWUpdate framework
- Explore Uptane's method to verify the images using the SHA256 hashes and explore the methods used in the current implementation of Aktualizr for this purpose
- Incorporate these methods into the implementation of Aktualizr for SWUpdate or RAUC

II. Efficiently Installing the image and making further space and time-optimisations

- Explore the various fields in the sw-description that need to be present for our use case
- Make use of sw-description fields for signed images to pass the signature of the image that requires verification
- Pipeline the installing requests to handlers of various filesystems as used by the rootfs for that device
- Employ the installed-directly flag to install the image directly to the Standby partition

III. Configuring the bootloader to boot into the new partition on the next restart

- This consists of the final part of the pipeline, where I will have to configure the bootloader to boot into the Standby partition after the next reboot
- SWUpdate has functions written to configure this, I'll have to pipeline it to my desired function
- The last task is to verify the installed image and fallback in case of a faulty update

Implementation

The general implementation mechanism is discussed in detail in previous sections. Here, I'm going to talk about some specific

implementation challenges and features that will make the project more compatible and space efficient.

Uptane::Target

In cooperation with the mentors, I discussed possible ways to define *Uptane::Target* for A/B partition-based updates using either RAUC or SWUpdate. The task at hand is to report the hash of the Target image. As both the package managers have similar metadata formats (sw-description and manifest) and both contain the hashes of the signed images, we decided to take a generic approach.

```
software =
{
        version = "0.1.0";

        hardware-compatibility: [ "revC"];

        images: (
                {
                    filename = "core-image-full-cmdline-beaglebone.ext3";
                    device = "/dev/mmcblk0p2";
                    type = "raw";
                    sha256 = "43cdedde429d1ee379a7d91e3e7c4b0b9ff952543a91a55bb2221e5c72cb342b"
                }
        );
        scripts: (
                {
                    filename = "test.lua";
                    type = "lua";
                    sha256 = "f53e0b271af4c2896f56a6adffa79a1ffa3e373c9ac96e00c4cfc577b9bea5f1"
                 }
        );
}
```

This image shows what a typical sw-description would look like for a signed image. *Example is borrowed from SWUpdate Documentation.*


```
{
  "signatures": [ ... ],
  "signed": {
    "_type": "Targets",
    "expires": "2038-01-19T03:14:06Z",
    "targets": {
      "targets/file.txt": {
        "custom": {
          "ecu_identifier": "01:02:03:04:05:06",
          "hardware_identifier": "abc-def",
          "release_counter": 1,
          "sw-description": {
          }
        },
        "hashes": {
          "sha256": "fbf121c8e85875ffb9d50eab2cfa2a692df0e3e06afc2913492abe128bae66d0",
          "sha512": "ac51637364cff0f6802e087c8c6a51f1925384af639214282a67e093495ef746d8f1c2326cf7182a1067fd69f6049247
        },
        "length": 12
      }
    },
    "version": 1
  }
}
```

The above image is taken from a Team Discussion. The idea is to embed the information provided in metadata inside the *custom* field of the current *Uptane::Target JSON object.* This would ensure that we are not making any changes to the current definition of Uptane::Target, thus ensuring **backwards compatibility.**

Direct Streaming

The issue:

Embedded devices have very limited storage capacity. On top of that, for A/B partition based updates, we need to have two root partitions thus consuming more space initially. Also, the image to be downloaded is considerably large and thus would consume a lot of space when initially downloading it onto the active partition. This would require extra space requirement for both A-B partitions.

The solution:

Instead of downloading the image on the active partition we can directly stream the image to the inactive partition (the one on which it is supposed to be installed). This would save the storage space on the active partition, as well as the amount of required copying time, or number of cycles needed to copy the image from active to inactive partition) thus leading to more efficient time and storage utilisation.

This method, called direct-streaming,is a feature provided by both SWUpdate and RAUC, and will be integrated into the project for above reasons.

Delta Updates

The issue:

The size of images are steadily increasing as new features are added on each update. Besides, many times, the updates are just minor bug fixes or add only a few small features. In such a case, the new image

might contain only minor changes in the files which are very small in size compared to the full image. Bandwidth is not cheap and thus downloading the whole image on an update would not be an ideal approach as it will consume more time and money. Besides, we would need a large amount of storage space for downloading the image as well.

The solution:

The efficient way out is to just download the differences between the current image and the new image thus saving the bandwidth required to download the full image. This strategy, termed delta-updates, can be implemented using several available algorithms.

Both SWUpdate and RAUC support delta-updates and incorporate the most suitable algorithm as per their framework for implementing this feature. I will be exploring the APIs provided by the respective package manager and the mechanism they employ to implement the delta-updates features into aktualizr.

Timeline

Pre-Selection Period

- Reviewed the framework of Aktualizr and tried to understand the task at hand
- Discussed ideas with mentors on how to proceed with the project, delving into the details of any potential problems that could occur during implementation
- Analysed the SWUpdate and RAUC features and discussed the potential scope of the project and some basic structural parameters of the ‘Target’ image under the SWUpdate framework

Weeks 1-2

- Understand the security framework of Uptane Client
- Learn how Uptane verifies an image to be downloaded
- Choose either SWUpdate or RAUC to implement in the project

- Decide what a Target JSON object looks like

Weeks 3-4

- Working within the constraints of the framework discussed above, propose the most suitable strategy to embed sw-description into the current definition of the Target
- Successfully download an image into the A partition without verification

Week 5

- Work on the verification of the downloaded image using the hash provided during in sw-description
- Design the sw-description parser (if needed)

Weeks 6-7

- Pipeline the installing request to the respective handler of the concerned filesystem for installation into the inactive partition
- Configure the bootloader to load the inactive partition as the rootfs on the next reboot
- Review the current implementation of direct streaming in SWUpdate’s framework

Weeks 8-9

- Work on the post-installation verification of the updated partition
- Pipeline the rollback mechanism for detecting a fault in an update as per Uptane’s policy
- Implement the hardware watchdog to configure the bootloader to boot into the old partition in case of update failure

Weeks 10-11

- Work on implementation of direct streaming to facilitate direct updates
- Delta-updates (extended feature)

Week 12

- Testing, code cleaning, commenting and documenting
- Writing the Project Report
- Merging the final implementation into aktualizr’s official repository

Deliverables

The following deliverables will be provided by the end of the project:

1. A project report detailing every aspect of the implementation and design choices made
2. Upstream an implementation of a new package manager for A/B partition-based devices, as discussed in the Objectives, to aktualizr's repository

My Motivation for Working with the Linux Foundation

My first interaction with operating systems was due to a *'Filesystem check failed'* error that occurred during the late summer of '21 while booting up on my older laptop. Although I could troubleshoot the error, I was still trying to understand why the error occurred when I realised it was due to a power failure. After further digging, I learned there was an inconsistency in the disk mapping of allocated blocks and inodes being written at the time of the power failure. This led me to think about why such a check was needed in the first place, how writes are performed, and what exactly these blocks and inodes are. Learning about the design of FileSystem instilled a deeper appreciation of the design choices and got me wondering *how to build systems that are more efficient and, at the same time, secure and robust to failures.*

From then on, I followed a focused path and started learning about computer systems through structured coursework in **Digital Electronics, Computer Organisation, Operating Systems, Linux Kernel Programming** and **Parallel Computing** during my 3rd and 4th years. Through these courses, I learned about the **MIPS** architecture and implemented a custom-designed processor and ISA using Verilog. It was a challenging task that included critically examining possible errors and building up hardware logic for complicated instructions like jump and branch. A project on implementing **CoW** (Copy-on-Write) in GemOS, a lazy memory allocation scheme for processes, gave me a lot of insights and a practical understanding of paging.

To further my interest, I’m currently examining *the possible ways to reduce the data handling overheads caused during chained invocations in serverless (FaaS) systems*. The task is to analyse the architecture of Apache OpenWhisk and explore various novel mechanisms and filesystem optimisations that can be adopted to reduce such overheads.

These courses provided a strong foundation, but a GSoC project with the Linux Foundation would be the bridge to the real world. The following are my key motivations for working on the Uptane project:

- **Real-world Projects, Real Impact:** I'd get to contribute to a mainstream project that is part of the Linux Foundation and the broader open-source community. My code would have a tangible impact on real users in the vast embedded systems industry and not just remain theoretical.
- **Learning from the Best:** Working alongside experienced developers would be an incredible opportunity to gain practical coding experience and insights. They'd mentor me through the development process, exposing me to the realities of OS development and best practices.
- **Open Source Collaboration:** GSoC provides a platform for introducing new developers to open-source projects. I'd be able to smoothly onboard and collaborate with a global community of developers, gleaning from their expertise and contributing my own knowledge.

GSoC is more than just a summer program; it's a springboard for my OS career. As a student, I aim to pursue **post-graduate** work in a related field, which I’m eagerly looking forward to. This project will prove to be a great opportunity to work on a large scale problem in the domain of Operating Systems and thus give me an idea of what I’m going to face in my future life. By combining my academic grounding with the practical experience and mentorship offered by fellow contributors and mentors, I'll be well-equipped to tackle the challenges and opportunities that lie ahead in the ever-evolving world of operating systems.

What excites me to be working with Uptane

My passion for operating systems goes hand-in-hand with my interest in electronics and embedded systems. During my time at IIT Kanpur, I actively participated in the Electronics Club, eventually becoming its coordinator. This role allowed me to lead a team of enthusiastic students in exploring cutting-edge electronics and building exciting IoT applications.

I gained extensive experience working with various microcontrollers and microprocessors through these projects. This included popular platforms like Arduino (ATmega328P), NodeMCU (ESP8266), and Raspberry Pi 3. One particularly notable project involved a touch-based projection system. We used cameras to detect user touch on a projected screen, translating gestures into left-click, right-click, or drag commands for the computer. This project was entirely implemented on a Raspberry Pi and Intel D435i camera.

In essence, working with Uptane allows me to combine my hands-on experience with embedded systems and my interest in the low-level programming of operating systems. This project aligns perfectly with my academic background and career aspirations, making it a truly exciting opportunity.

Why am I an ideal candidate for the project?

My experience makes me a strong candidate for the Uptane GSoC project. Here's why:

Technical Skills and Real-World Experience:
My undergraduate coursework equipped me with a deep understanding of Linux systems concepts. This foundation, combined with my proficiency in low-level C/C++ programming, positions me to dive into the Uptane codebase and grasp its intricacies quickly. Furthermore, my summer internship at **Uber** was a masterclass in navigating and comprehending large codebases. There, I gained invaluable experience in designing, testing, and putting a tool to use in production. These are all skills that will be instrumental in this project.

Passion that Goes Beyond GSoC:
My enthusiasm for operating systems isn't limited to this GSoC opportunity. It's a genuine passion that fuels my involvement in other areas. I've undertaken several challenging projects in the systems domain (refer to previous section) in which I was able to successfully apply my knowledge to achieve tangible results. This drive to learn and contribute is a core part of who I am.

Proven Ability to Deliver on Large-Scale Projects:
My ability to tackle complex projects is further validated by my success at Uber. Not only did I complete my internship project, but it also earned me a Pre-Placement Offer (PPO). This demonstrates my capacity to work effectively within large-scale teams and projects and deliver impactful results.

You can refer to my resume for further information about my projects and achievements so far.

Time Commitment

I will be able to commit **35-40 hours** every week and my work timings will be pretty flexible. Apart from the GSoC project, I have a commitment to complete my Undergraduate Research Project. This is an extended project which will be wrapped up by mid-May 2024, after which the GSoC project will be my only commitment.

References

1. A/B (seamless) system updates | Android Open Source Project
2. Get-Started-With-Uptane - Jon Oster
3. Uptane Documentation
4. Discord and Zoom discussions with Project Mentors
5. SWUpdate Documentation
6. Abhijay's 2023 Uptane GSOC proposal
7. RAUC Documentation