ARED - Yocto Project Architecture v1.0.3

- Changelog
- Intro
  - Conventions
- General Architecture
  - Project’s goal
  - General system architecture
- General Architecture
  - Operating System stack
- Base Operating System
  - System partitioning
  - SSH connection
  - Firewall
  - Time synchronization
  - Dockerized client services
  - Dockerized client services
  - VPN connection
- Cloud connection
  - Content management
  - Future development
    - Content management summary
- System monitoring
- Device provisioning
  - Provision procedure
  - Provision procedure
- System update
  - Dual image approach
  - Update flow
  - Update strategies
  - Containers update
- Summary of the project architecture
  - Base operating system
  - Cloud connection
  - Device provisioning
  - System update
- Implementation plan

Changelog

- v1.0.3 - 13/09/2022
- Change provisioning process description
- v1.0.2 - 28/04/2021
  - Expand content management options
- v1.0.1 21/04/2021
  - Update OS stack diagram
  - Explain the need for OpenVPN and openSSH in the system
  - Add a time synchronization section
  - Add a content management option with the *rsync* tool
  - Ad content management summary
  - Mention the possibility of scheduling system updates
  - Propose to use *rsync* as a content management tool
- v1.0.0 - 15/04/2021
  - Initial document release

Intro

The goal of this document is to formulate technical requirements based on the business requirements received from the client. The possible solutions will be researched and described there.

Conventions

In the case of commands

- *$ command* - command to be executed on the host PC
- *# command - command to be executed on the target device*

In the case of log/code snippets

- (...) - part of the log was truncated for readability
- \ - the content of the line moved to the second line (line break) to fit on the page

General Architecture

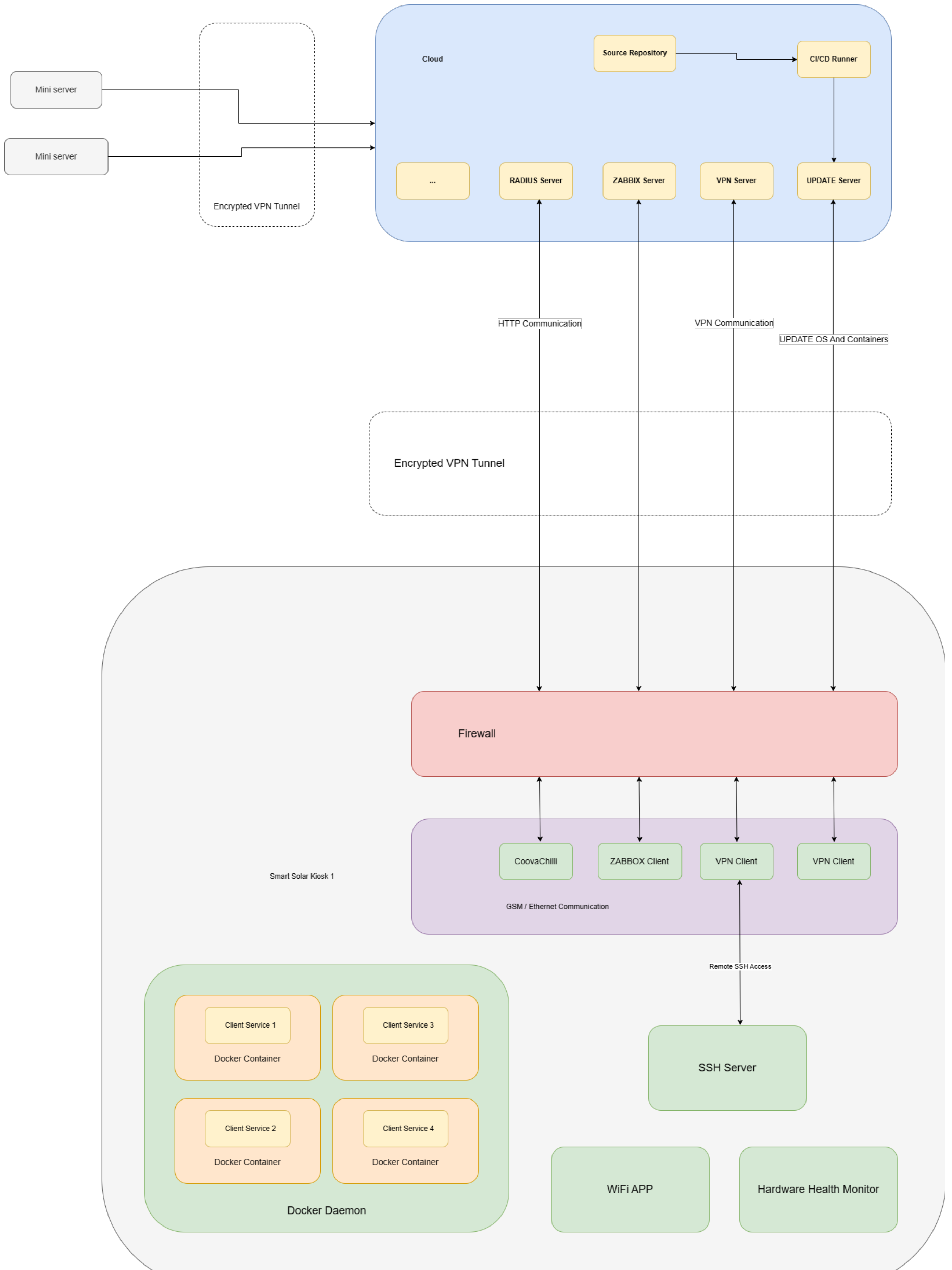
Project's goal

The main goal is to develop an edge device called ARED Mini Server (also known as Smart Solar Kiosk), which will provide multiple services to the local community. It was meant to replace internet connection in places where the global medium is unavailable due to lack of infrastructure or financial costs.

The main idea is to allow end users to download multimedia files, documents,

and Android applications without direct internet access, but only by connecting their smartphones to the access point provided by Smart Solar Kiosks.

General system architecture



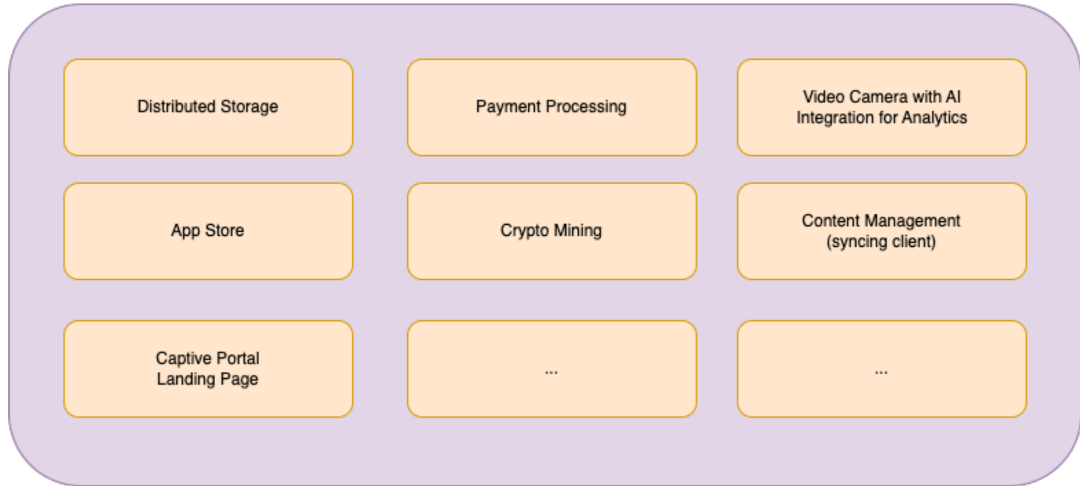
Mini server
Mini server
Encrypted VPN Tunnel
Cloud
Source Repository
CI/CD Runner
...
RADIUS Server
ZABBIX Server
VPN Server
UPDATE Server
HTTP Communication
VPN Communication
UPDATE OS And Containers
Encrypted VPN Tunnel
Firewall
Smart Solar Kiosk 1
CoovaChilli
ZABBOX Client
VPN Client
VPN Client
GSM / Ethernet Communication
Remote SSH Access
Client Service 1
Docker Container
Client Service 3
Docker Container
Client Service 2
Docker Container
Client Service 4
Docker Container
Docker Daemon
SSH Server
WiFi APP
Hardware Health Monitor

YOCTO PROJECT OPERATING SYSTEM

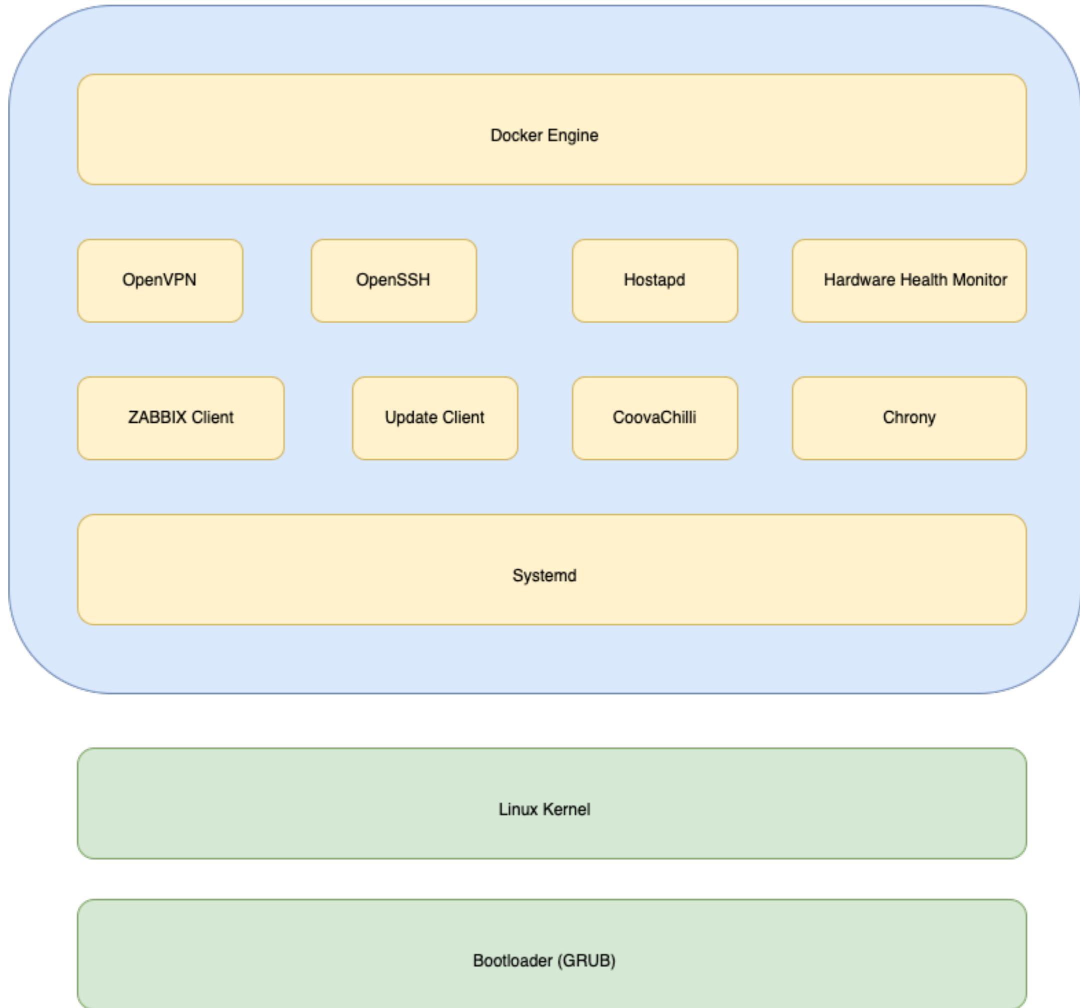
General Architecture

Operating System stack

Containers



Base OS



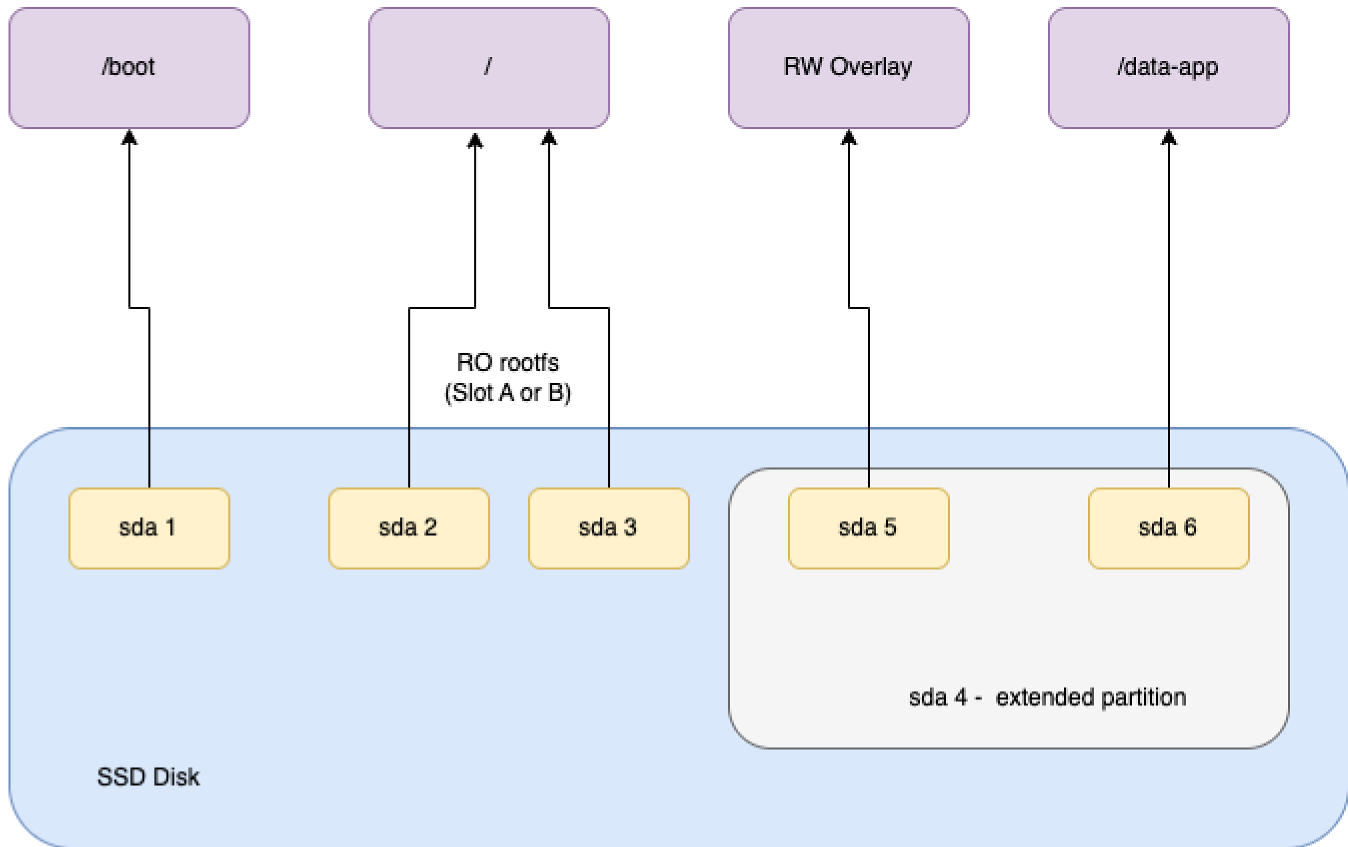
Base Operating System

This section covers the base operating system components.

System partitioning

The system is divided into five partitions taking into account the dual image approach used in the update system. A short description is provided below along with the name of each partition.

- *boot - first partition; includes firmware and other files necessary to run the platform*
- *rootfsa - second partition; first slot for the system*
- *roofs - third partition; second slot for the system*
- *rwoverlay - fourth partition; used for read-only rootfs overlay*
- *data-app - fifth partition; includes space for user files and other data*



SSH connection

One of the critical requirements for the implementation of the project should be the ability to establish an SSH connection with any Smart Solar Kiosk located in the field.

Ultimately Mini Server devices will be scattered all over the continent, thus it will be impossible to manage and control them without the remote control. Moreover, SSH connection can be available only through a VPN tunnel to isolate from local WiFi-connected devices and improve security.

To improve the security even more, every Mini Server should have its own private key generated that is mandatory to connect via SSH. Connection using a password should be prevented.

The ability to log into any Mini Server will be possible only when all devices are inside one network, which will be ensured by the implementation of a VPN in the system.

Firewall

Appropriate firewall configuration is aimed at increasing the security of devices in the field. We suggest that only a minimal set of inbound and outbound connections be allowed.

A list of all possible connections needs to be prepared, and the system should block each one that is not allowed.

Time synchronization

Ensuring the current time on the Mini Servers will enable the correct implementation of actions to be run at the appropriate times. An example of such actions may be checking for updates only during the hours when the device is not used. The correct time on the Mini Servers can be ensured by installing and properly configuring chrony.

Dockerized client services

To ensure the optimal environment for every service without any dependency conflict, Docker containers infrastructure can be used. Every service can be encapsulated into a separate Docker container which can contain only necessary dependencies.

According to the provided materials, the following services can be dockerized

- Android applications store
- Video camera with AI integration for analytics
- Payment processing
- Distributed storage
- Crypto mining
- and more

Target system requirements and architecture have very much in common with the balenaOS which is an operating system targeted at running Docker containers on edge devices. The system is open source and built using Yocto. It can be customized and rebuilt by oneself if needed.

But the balenaOS system also has some major issues like

- now officially supported mechanism to update the base OS
- generally poor support for image customization on the build phase; the officially supported method for updating base OS only allows updates to the official images built by the *balena* team; otherwise, some hacks are needed

Dockerized client services

To have much greater flexibility over the OS, a custom one can be developed(using Yocto). Even if not using the *balenaOS*, some of the good concepts or components can be integrated there.

For example, the balenaEngine is advised to be used (no matter if using the balenaOS or the custom image). The first advantage is ~4 times smaller binary than the original Docker (decreased system image size). The second advantage (in terms of the update system) is the support for c ontainer deltas.

balenaOS components and *meta-balena* Yocto metadata are released under the *Apache 2.0* license, which does not prevent them from integrating into proprietary products.

This section contains information about services that should be run on the cloud server in order to fulfill the architecture requirements.

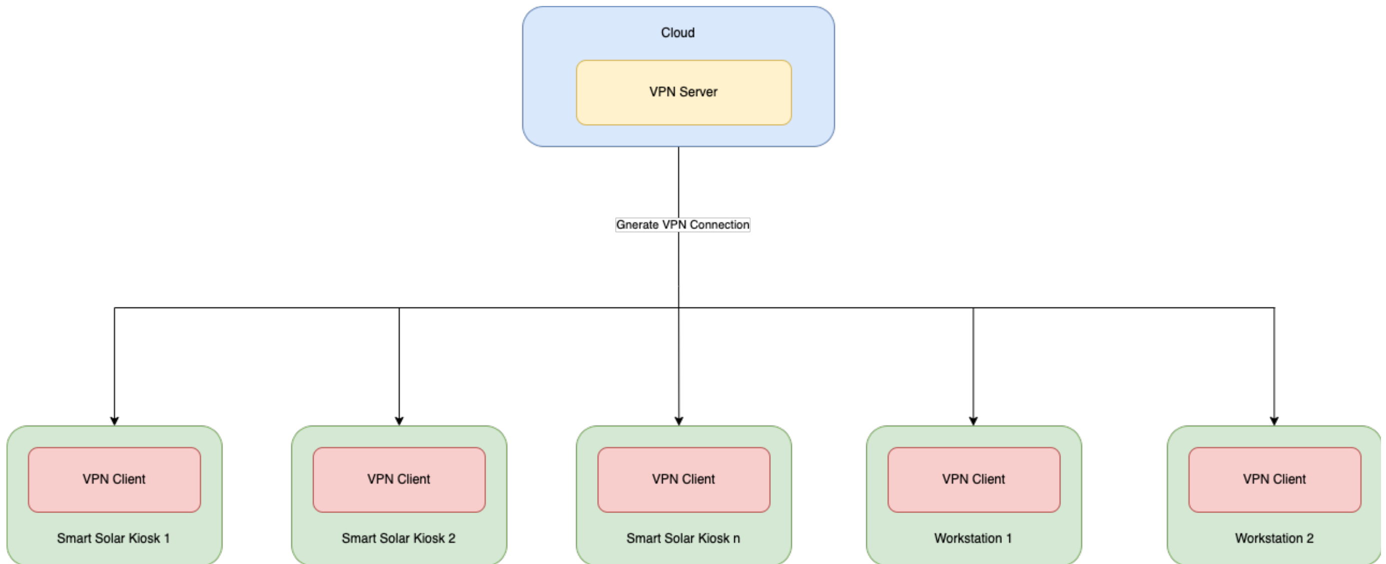
VPN connection

Access to any Mini Server should be provided through a VPN connection. For this purpose, a VPN server should be started on the cloud, which can be a service run inside the container.

A configuration file should be generated for each Mini Server when the device is first started. It will be used by a VPN client installed on each device.

Moreover, such a configuration file can also be generated for workstation operators, thanks to which they will have access to all devices in the VPN network.

Adding a VPN to the system architecture ensures the creation of a network layer thanks to which all elements (cloud, Mini Servers, workstations) can be seen and available to each other. The fulfillment of this condition is required to be able to remotely log in to Mini Servers using SSH. This ensures the ability to perform diagnostic and repair work on devices.

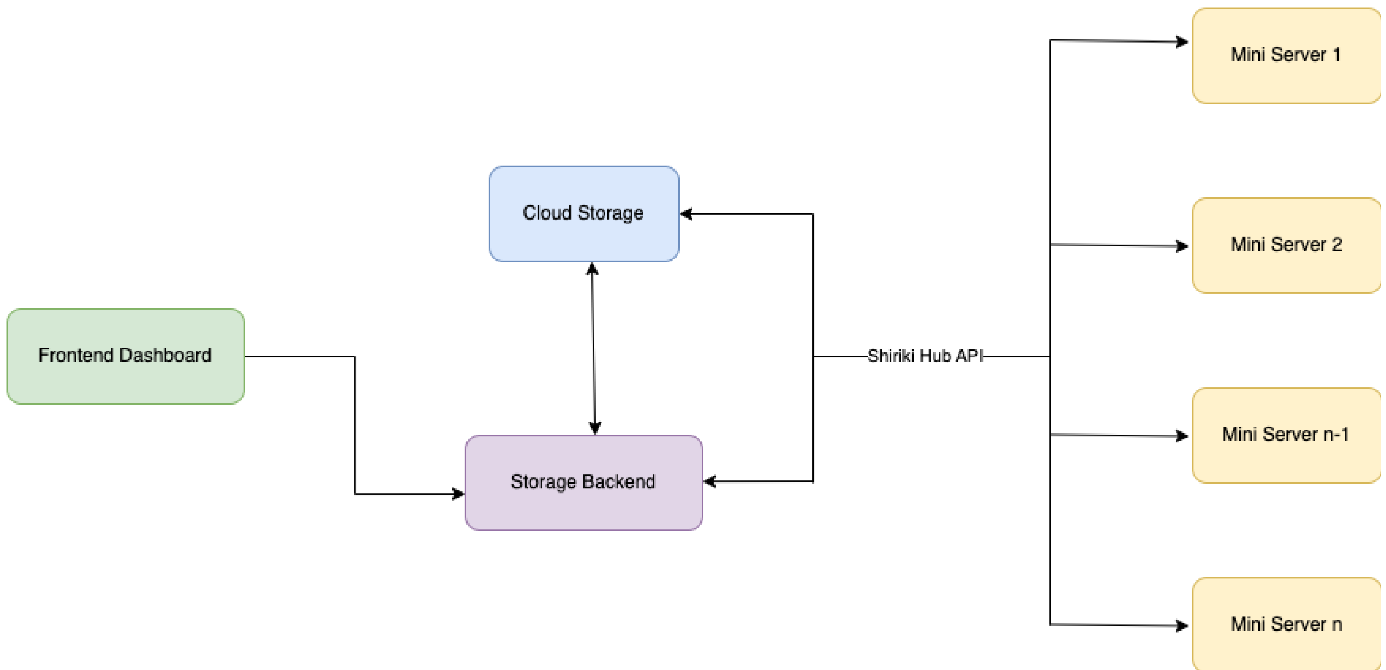


Cloud connection

Content management

The ARED Content Manager (referred to as *CM*) shall provide an endpoint for ARED edge devices (Mini Servers) to download content (Android apps, documents, music, videos, etc.) for local distribution. It shall provide a standard HTTP response code, verbs, and authentication.

According to the provided materials, the asset storage should be done using Google Cloud Storage Buckets or equivalent hosting services such as AWS S3 or Azure Blob Storage. The backend services shall manage which assets are exposed to which device.



The *CM* shall use API keys to authenticate requests. Each Mini Server should have its own unique API key that shall be provisioned by its configuration client. Restricted API keys can be used for granular permission to content in conjunction with configured device realm.

Content management will include ads, surveys, and offline files. The first step will be to implement offline files support, the rest could be prepared in a similar way.

In the proposed Shiriki Hub API we saw some inconsistency. The provided materials indicate how assets should be described in cloud storage.

```
{
  "id": "0000234",
  "name": "ared_welcome.ts",
  "active": true,
  "realms": [],
  "created": "2021-01-18 09:01:00:000",
  "md5sum": "xxxxxxxx",
  "version" : "0000001",
  "url": "asset url",
  "description":  null
},
```

But we see some differences in the actual examples given in the file list. We have an example of an available video there described as follows:

```
{
  "id": 4,
  "category": "media",
  "sme": null,
  "created": "2021-03-30T10:35:12.712473+03:00",
  "updated": "2021-03-30T10:35:12.712510+03:00",
  "file_type": "Video",
  "file": "https://storage.googleapis.com/m-shirki-production/files/rea
/lms/8/media \
           /Video/76567e5a-f237-49e3-8671-b6d661f61bb5.mp4",
  "unique_id": "76567e5a-f237-49e3-8671-b6d661f61bb5",
  "status": "Enabled",
  "realm": 8,
  "realm_name": "EDIT Africa Solar kiosk"
},
```

The main missing field is the one containing the value of md5sum of the given file. Its absence makes it impossible to verify the correctness of the file download.

Another alarming issue is the *Content Client* installed on Mini Servers. It should be responsible for exchanging files with the cloud, checking the correctness of downloading files, and verifying the list of files that should be on a given Mini Server, depending on the region (realm) to which it has been assigned. In this scenario, the *Content Manager* shall be the source of truth for all assets.

According to the provided materials, *Content Client* should use a small local database (such as SQLite) to store a list of files located locally on the device. This list should be sent to *Content Manager* after each successful download of the file by the *Content Client*.

We strongly advise against using this approach as it is not well scalable. From the experience gained while working on another project, we know that the size of a file containing a similar list saved in JSON format can grow quickly. 50,000 entries generate a file with a size of about 15 MB. Additionally, the necessity of sending this list after each successful download of a new file seems unacceptable. This would lead to situations where downloading a file with a size of several KB would trigger the action of sending a file with a list having a size of several times larger (for example several dozen MB). When downloading a few such small files in a short time, it would result in sending the file list several times. Such a process would unnecessarily burden the Mini Server System and use the available transfer, which is very limited.

Content management must be implemented using Google Cloud Storage. In addition, file transfer must be resistant to poor internet connection due to the conditions in which the Mini Servers will work. This means that the target solution must meet the following requirements

- possibility of further development of a given solution
- possibility of resuming interrupted transfers
- resistance to breaking the connection
- basic sync feedback on the synchronization status - information about the status of the synchronization for a given device
- full sync feedback on the synchronization status - view the list of files available on the device

- cooperation with Google Cloud Storage as a storage backend

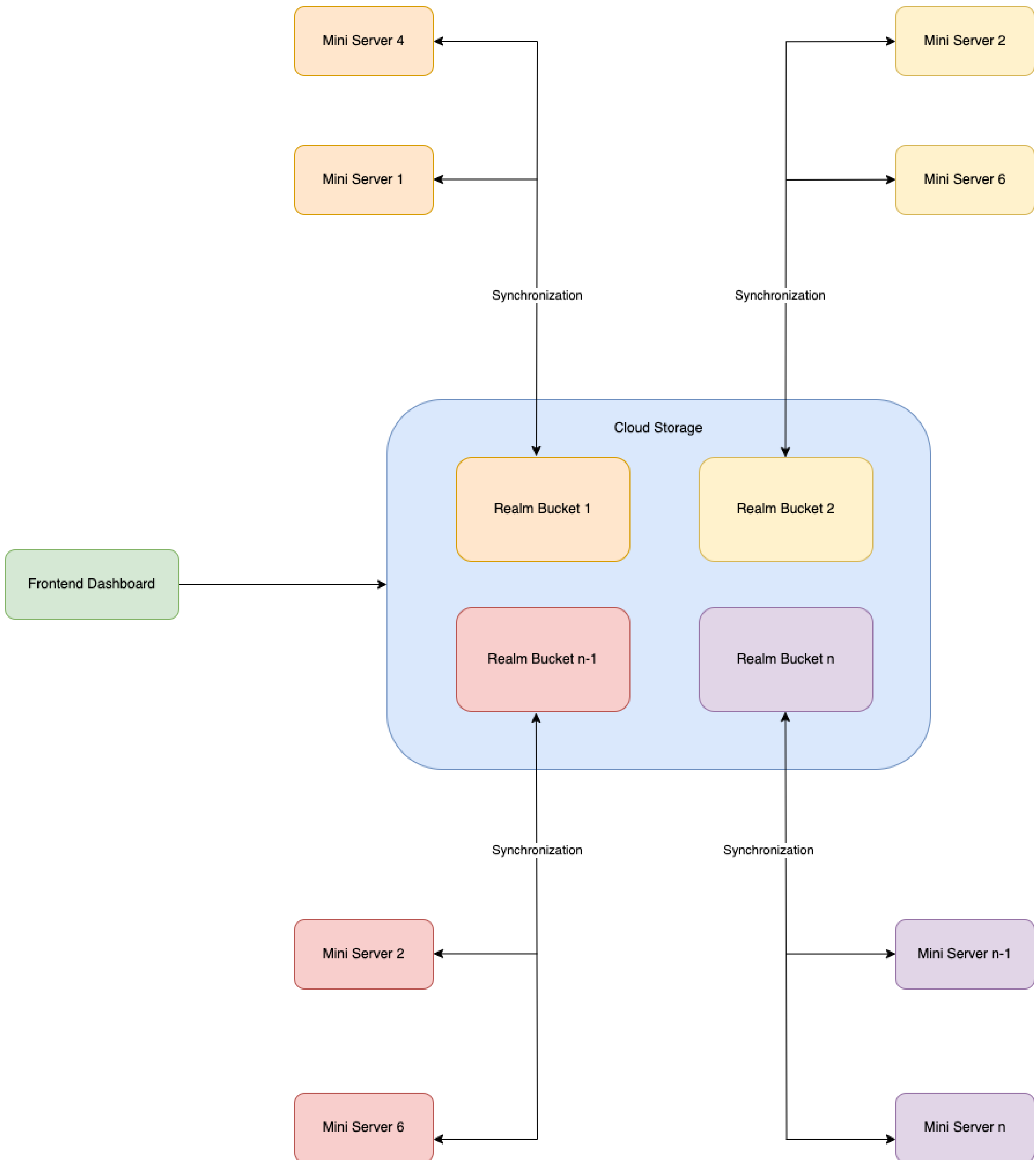
In the further part of the content management section, three possible implementation options will be presented. Below is a table with information on whether a given solution meets a given requirement.

	Solution	Further development	Resuming interrupted transfer	Resistance to breaking the connection	Basic sync feedback	Coop with CGS
1	gsutil sync	YES	NO	NO	NO	YES
2	combination of gsutil rsync and cp	YES	YES	NO	NO	YES
3	custom solution from scratch based on Shiriki API	YES	YES	YES	YES	YES

The first option to implement content management is to use Google CloudStorage buckets with the gsutil tool.

The factor deciding which file should be on which device is the realm to which it was assigned. To meet these requirements, one possibility is to use cloud buckets (using Google Cloud Storage). Each bucket should be assigned to one region. When adding new files to the cloud, they should go to the appropriate bucket. Thanks to this approach, we are able to easily divide the content going to cloud storage.

The task of the Mini Servers will be periodic file synchronization. The source of truth here will be the bucket assigned to a given region, which means that Mini Server, when synchronizing, will have to delete unnecessary files or download the missing ones.



In the proposed solution, the role of Content Client can be served by the *gsutil*

- gsutil advantages
  - use created credentials in communication with the storage
  - provide rsync command which easily allows synchronizing the content stored on the given Mini Server with the content from the given bucket
  - does not need any external list of stored files
  - created in Python, a recipe for Yocto build is not available but can be generated from the repository, as it is developed under the *Apache 2.0* license, which does not prevent from integrating them into proprietary products

The access restriction can be also easily met in such an approach, by granting each device access to the files in the selected bucket (realm) only.

Such an approach greatly simplifies the implementation of the solution by using existing tools rather than implementing them from scratch. The backend to be developed here should be fairly minimal, while the client on the Mini Server would be a simple script using one of the CLI tools. This way the solution could be shipped to market much faster.

Unfortunately, this solution may not meet e.g the resumable download requirements but can be improved to do so in the second proposed option.

The second option is an extension of the first. Thanks to the use of additional logic, file transfer between the cloud and Mini Servers can be resistant to a weak Internet connection. In this solution, the *gsutil rsync* command will only be used to compare the content between the bucket and the Mini Server. By using the -c, option the file hash will be checked during the comparison.

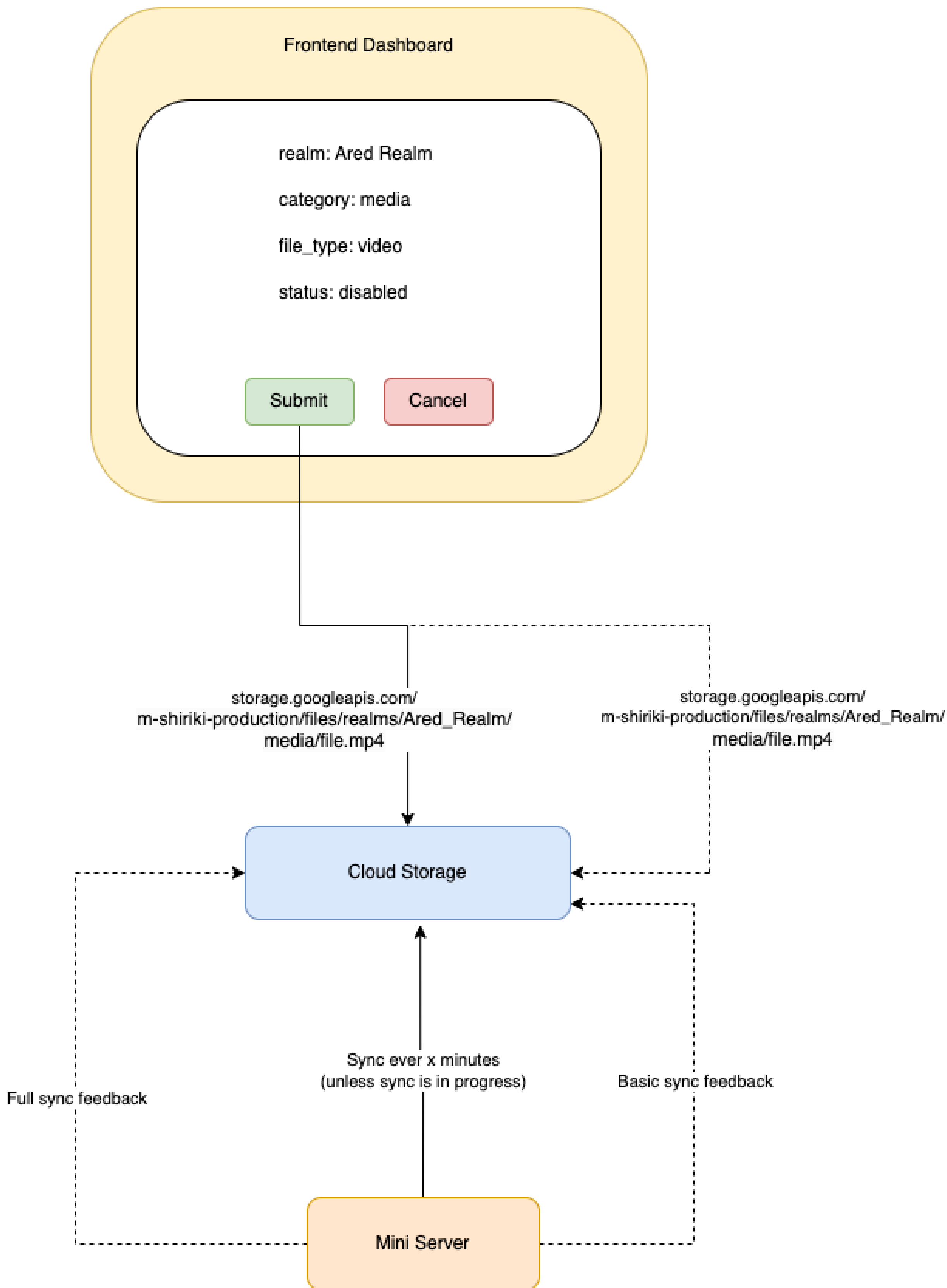
Based on this information, Mini Servers will delete local unnecessary files and the missing ones will be downloaded using the cp command. Using it provides resistance to broken connections and allows you to resume the download.

Future development

In the future, the option using the *gsutil* utility may be extended with the following functionality

- basic sync feedback with the status of the synchronization carried out by theMini Server - it will require adding Google IoT Core and opening communication with the Mini Server, for example via the MQTT protocol, the status can tell if a sync is in progress or when the last successful sync was completed
- full sync feedback - at the operator's request a given Mini Server could send a list of stored files, such a list would also be sent via the *MQTT* protocol
- preparing the structure of folders in buckets that will reflect the information proposed in the custom Shiriki API, i.e. category, file type, realm, status, or information related to SME, thanks to which it will be easy to distinguish, for example, the status of a given file (Enabled / Disabled)
- file synchronization performed according to a set schedule

The diagram below shows content management after implementing the second option, the dashed arrows indicate possible system development in the future.



The last option involves the use of Shiriki API for the implementation of content management. This approach will be the most time-consuming as it will require custom development almost from scratch. The solution would be based on [libraries enabling communication with Google Cloud Storage](#).

The proposed API will require several changes, including the differences described on page 16. In the beginning, work will have to be focused on proper communication with Google Cloud Storage. Then it will be possible to add more functionalities. Firstly, basic ones such as downloading

files from storage, and comparing the hash of files to determine which to download. Many of these functionalities are already properly implemented in the *gsutil* tool we propose to use in the previous sections, but the custom Shiriki API option provides the greatest flexibility in implementing the target solution.

Content management summary

While analyzing the possibility of implementing content management, open-source solutions enabling the use of the current Shiriki Hub API were checked. Unfortunately, none of them met all the requirements. Using a custom API would require the development of a new solution. Such an approach would greatly delay the product's entry into the market.

We propose an implementation with the use of *gsutil*. Thanks to it, we are able to easily provide a synchronization system resistant to problems with the Internet connection. In addition, this solution may be developed in the future to provide additional functionalities, such as reporting the synchronization status for sending a list of files on the Mini Server data.

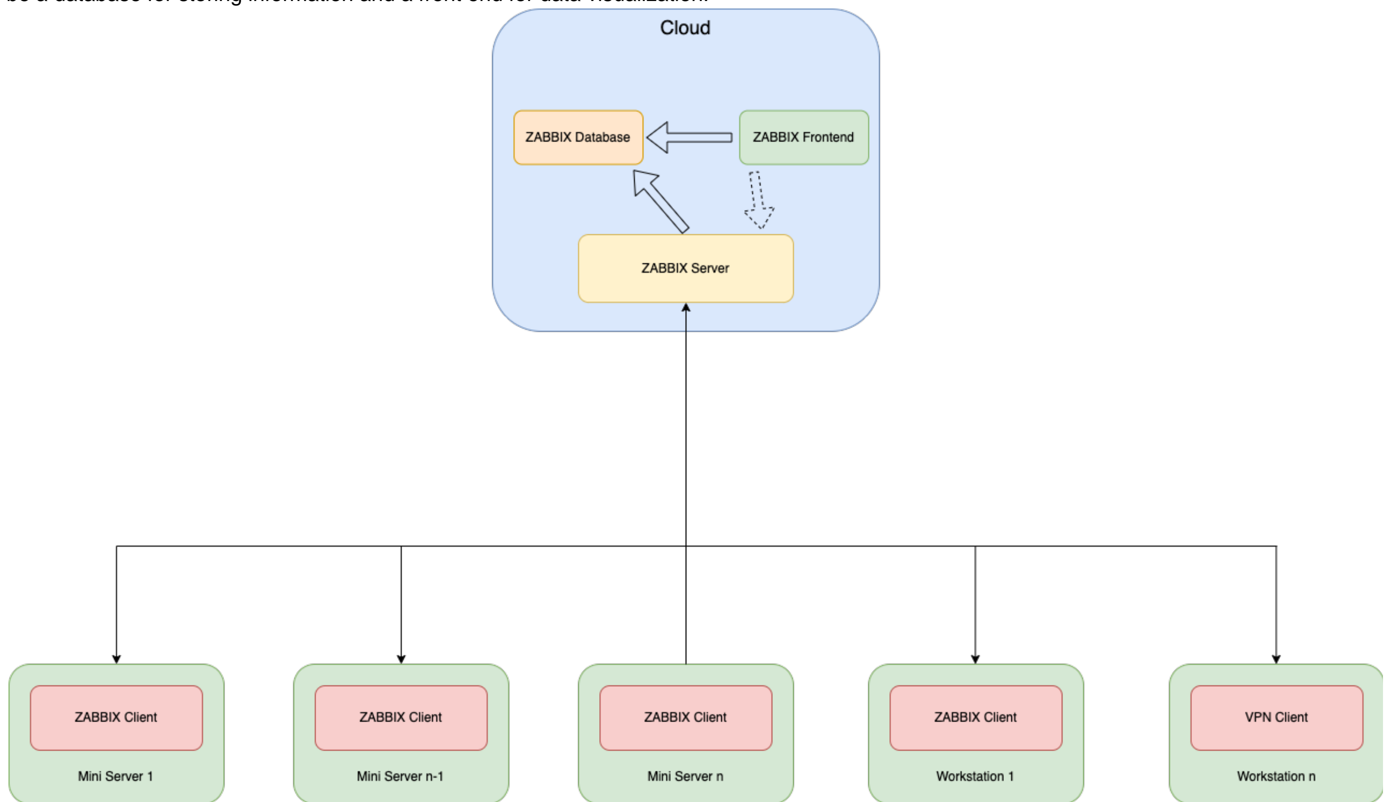
System monitoring

Monitoring of the entire infrastructure should be a very important element of the discussed architecture. Its proper implementation makes it possible to detect any existing problems, which then leads to their diagnosis and repair in a satisfactory time.

One of the available solutions for the above is [Zabbix](). Zabbix is used to collect, analyze and visualize data. It has several reasons to use it

- it is released under the GPLv2 license
- provide a huge amount of monitoring methods - various protocols can be used including SNMP, SSH, TELNET, HTTP/HTTPS, etc.
- stable software - Zabbix has LTS (Long Time Support) version

This solution requires a server installed on the cloud and a client on each device that should be monitored. On the server side, there should also be a database for storing information and a front end for data visualization.



Device provisioning

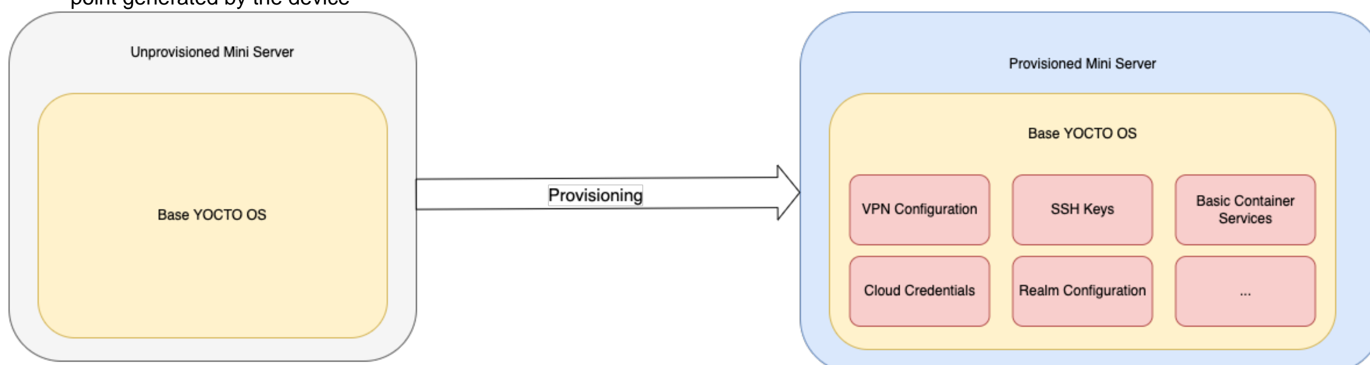
Provision procedure

After flashing the new image onto the Mini Server, we can say that it is in *factory* mode. At this stage, the provisioning procedure should be carried out to prepare the device for operation in production mode.

The result of such a procedure should be

- creating a VPN configuration and installing it on the device

- generating SSH connection keys
- cloud storage credentials
- device configuration for a given realm consists of elements such as
    - setting the language of the applications appropriate to the given region
    - assigning the Mini Server to the appropriate cloud storage directory for the proper implementation of the content management system
    - setting the configuration of services run on Mini Servers appropriate for a given region, for example setting the right name of the access point generated by the device



Provision procedure

The effects of performing the provisioning procedure should be saved on the partition that is not upgradeable. In the case of the discussed system, it will be the *overlay* partition. The configurations saved on it should be permanent, only deleted at the request of the operator, who should be able to perform an action called *factory reset* on the device. After doing so, the Mini Server should be in the *unprovisioned* state again.

Starting from release v0.11.0 of the Mini Server system image, the process is fully automated by the usage of installer-image. It is a special image designed to automate SSD flashing and the target system provisioning process. It uses the special script which

- flashes target image to the internal SSD storage
- call *provisioning* endpoint of provisioning service which is running on the **yocto-one** virtual machine
- decompress downloaded packages and place secrets on internal storage
- test connection with VPN (**yocto-one**) and Zabbix server (**yocto-two**)
- adds a new host on the Zabbix server with its IP address and name *mini-server_ID*, where *ID* is device unique ID that is calculated from the MAC address of one of the ethernet interface

The provisioning service itself, after authenticating the provisioning request creates a bunch of secrets for the given Mini Server which are:

- VPN configuration for VPN server that is running on the **yocto-one** virtual machine started on Google Cloud
- VPN configuration for Zabbix VPN server that is running on **yocto-two** virtual machine started on Google Cloud
- Google Cloud content management access key
- Google Cloud container registry access key
- Predefined values for country and *content management synchronization* interval which are **RW** and **10**

System update

The goal of this section is to show how the system update mechanism is implemented.

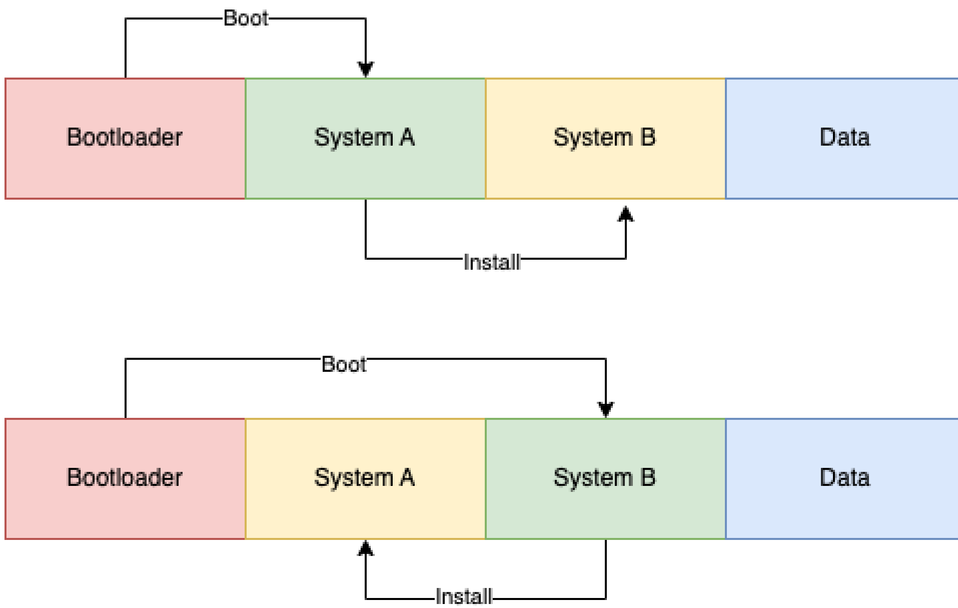
Dual image approach

The dual image approach implies having two partitions for the entire software running on the board. Synergy with the boot loader is often necessary because the boot loader must decide which copy should be started. Again, it must be possible to switch between the two copies. After a reboot, the boot loader decides which copy should be run.

This approach has several advantages

- installing updates on an inactive slot allows to run them in the background, which allows the system to continue working (minimal system downtime)
- in case of an incorrectly installed update or power outage when installing the update, we always have a system with an older version of the software, thanks to which the device is always usable
- the update installation is atomic which means complete success or no success

If system A is active, system B is updated. After the platform restart system B is launched and (if an update is accepted, e.g. no error occurs) it is a new active slot. In the case of the next update, system A will be updated.



Update flow

The below diagram illustrates the process of updating the installation and approving a new active partition.

Update strategies

- Manual update
  - an operator logs in via SSH downloads the relevant update image and executes
- Simple polling

    - the device regularly polls a given endpoint (e.g. HTTP URL) for a new update image - if there is a newer version available, it will be automatically downloaded and installed
- Update server with updated management dashboard
    - an operator can schedule updates via the dashboard
    - devices can also poll the server for the updates
    - the dashboard presents the versions present on the fleet of devices and the updated statuses

Advanced system update features

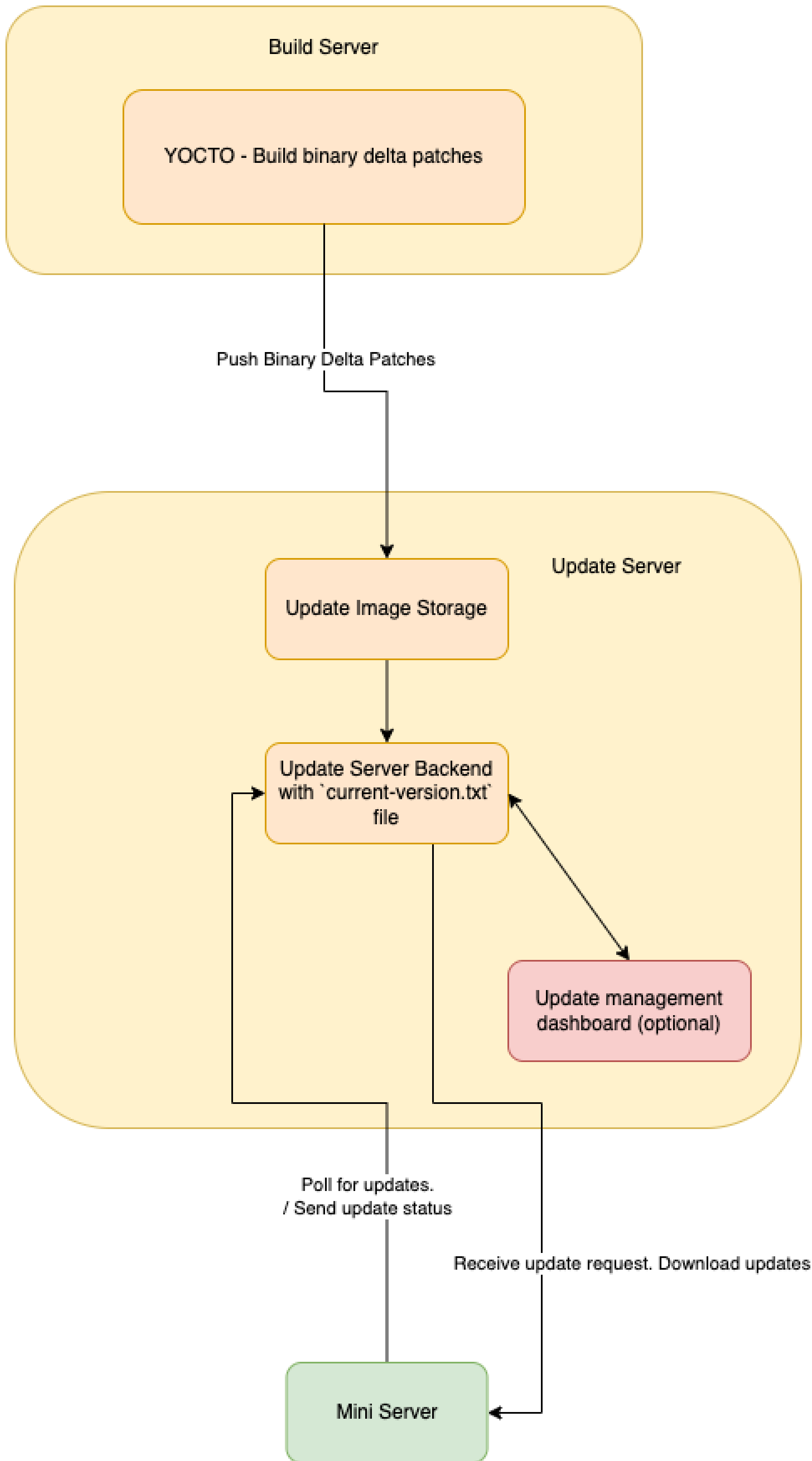
- Binary delta updates
    - significantly less bandwidth required to update the system
    - requires the *diff* files to be generated on the servers side
    - requires the *diff* file to be generated for each possible version combination - if we have many devices and many different versions in the field, the number of *diff* files to generate may grow over time
- Signed updates
    - only images signed by our private key will be accepted by the devices
- Encrypted updates
    - images can be encrypted to prevent interception and inspection by an attacker (if this is a threat in a given use case)

Update delta patches

At the moment, a *manual update* strategy is implemented in the system. We propose to expand it with the use of binary delta patches and the automation of the delivery of update files to the cloud server from which the update client will download them to Mini Servers.

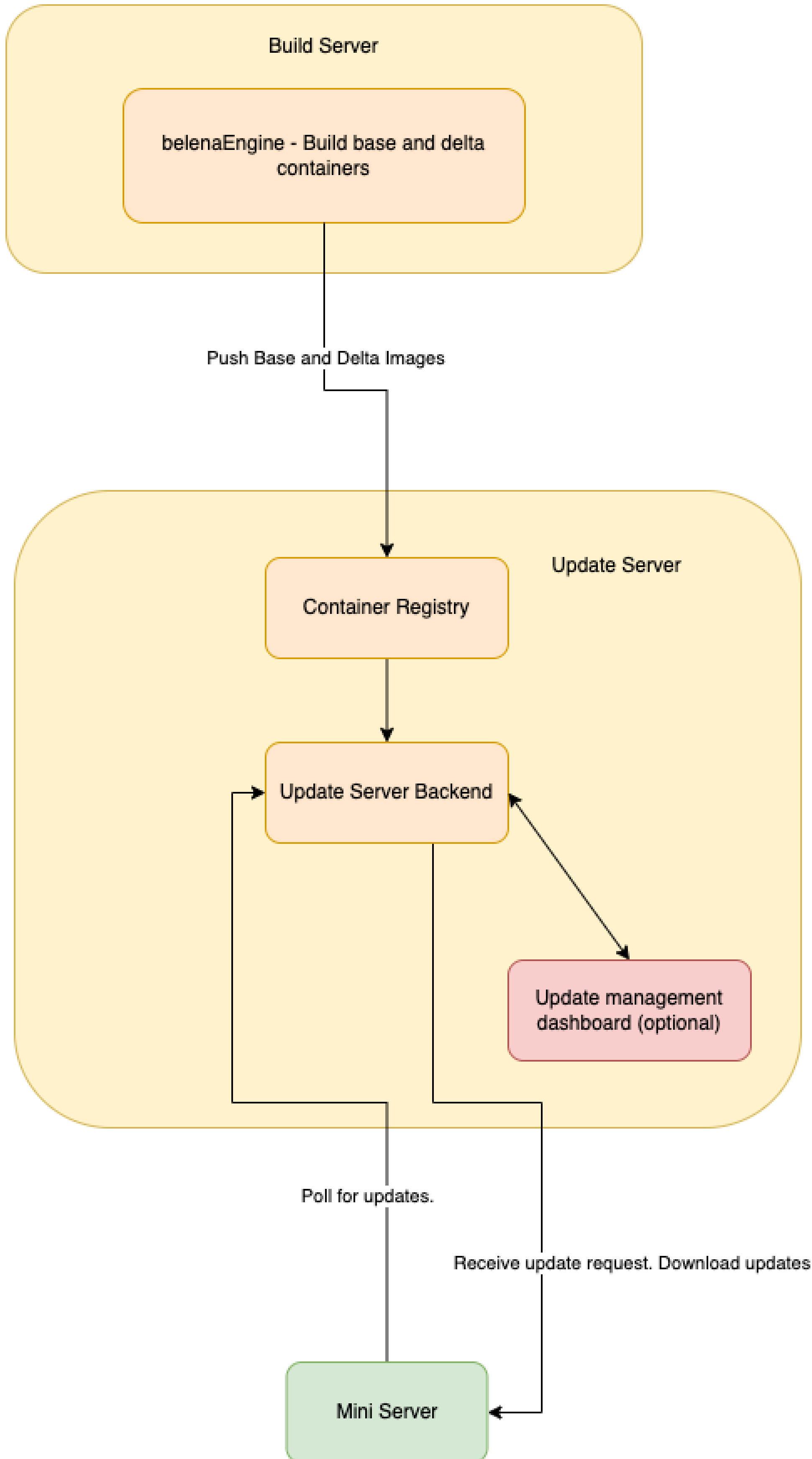
Cloud storage can be effectively used as an update file server. The simple automatic system can be implemented as follows. An appropriate folder structure must be created in the cloud. Adding the *current-version.txt* file with the description of the latest system version to the server will allow transferring of the entire logic of selecting the file to be updated to the application on the device.

The task of the updating application will be to periodically check the *current-version.txt* file and automatically start the update when a newer version of the system is available. Checking for available updates should be scheduled so that its hypothetical execution together with the required system restart does not interfere with the use of Mini Servers. Each device can return information to the server after the last update is performed. This information may include data such as the current system version or the status and date of the last update.



Containers update

The balenaEngine is advised to be used (no matter if using the *balenaOS* or the custom image). The first advantage is ~4 times smaller binary than the original *Docker* (decreased system image size). The second advantage (in terms of the update system) is the support for container deltas



Summary of the project architecture

Base operating system

- We suggest preparing a system that can run dockerized custom services
- It is advised to use a custom Yocto image with balenaEngine instead of *Docker* to run custom services inside containers. The advantages are:
  - less binary size
  - container delta updates support
- The prepared system should have an SSH server installed with a firewall properly configured

Cloud connection

- On the cloud there should be an OpenVPN server running, it can be started inside a container
- Adding a VPN server will allow connecting to devices in the field after proper configuration
- As a content management system, we suggest using specially prepared infrastructure on the server side and a simple script using the *rsync* to ol on the client side
  - there should be one user account on the server for each realm
  - *rsync* provides proper synchronization methods with resumable download possibility
- Zabbix should be used to monitor the whole infrastructure

Device provisioning

- The provisioning procedure should configure the device for a given realm
- Configuration should be stored on a persistent partition

System update

- The current update system should be enriched with additional functionalities
- Binary delta updates - correct implementation aims to limit the size of the update file
- OS update files can be uploaded to the cloud storage, and then it will depend on the updating applications on the Mini Servers whether the update process should be started
- Container images can be simply pulled from the registry. Using the balenaEngine delta update, the update size is considerably lower.

Implementation plan

The implementation plan - a list of the technical requirements and the effort of implementing each of them will be presented in a separate document.