D7.4 - FINAL BULK DEPLOYMENT TOOL

Deliverable ID	**D7.4**
Deliverable Title	**Final Bulk Deployment Tool**
Work Package	**WP7 – Deployment Toolchain**
Dissemination Level	**PUBLIC**
Version	**1.3**
Date	**2019-10-02**
Status	**Final**
Lead Editor	**Farshid Tavakolizadeh (FRAUNHOFER)**
Main Contributors	**Farshid Tavakolizadeh (FRAUNHOFER), Ákos Milánkovich, Balázs Kiss, Bálint Jánvári (SearchLab)**

Published by the CPSwarm Consortium

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731946.

Document History

Version	Date	Author(s)	Description
0.1	2018-08-01	Farshid Tavakolizadeh (FRAUNHOFER)	First Draft with TOC
0.2	2018-09-06	Farshid Tavakolizadeh (FRAUNHOFER)	Added requirements, background review, architecture and components
0.3	2018-09-10	Farshid Tavakolizadeh (FRAUNHOFER)	Added introduction, modified background review, architecture intro
0.4	2018-09-13	Bálint Jánvári (SLAB)	Added secure deployment considerations
0.5	2018-09-13	Farshid Tavakolizadeh (FRAUNHOFER)	Added implementation section, executive summary, conclusion, appendices, and annex
1.0	2018-09-23	Farshid Tavakolizadeh (FRAUNHOFER)	Addressed review comments.
1.1	2019-09-21	Farshid Tavakolizadeh (FRAUNHOFER)	Major updates in all sections (except background work) to reflect the final design and implementation.
1.2	2019-09-23	Ákos Milánkovich (SLAB)	Added certificate handling for Communication Library
1.3	2019-10-02	Farshid Tavakolizadeh (FRAUNHOFER)	Prepared for submission

Internal Review History

Review Date	Reviewer	Summary of Comments
2018-09-18	Davide Conzon (LINKS)	Accepted with minor comments
2018-09-18	Artiza Elosegui (TTTech)	Minor corrections
2019-10-01	Omar Morando (DGSKY)	Accepted with minor corrections
2019-10-01	Andreas Eckel (TTTech)	Minor corrections (a few typos only)

Executive Summary

This document is a deliverable of the CPSwarm project, funded by the European Commission's Directorate-General for Research and Innovation (DG RTD), under its Horizon 2020 Research and innovation program (H2020). It reports the results of "Task 7.2 Bulk deployment tools" between M13 and M33. This deliverable builds on top of "D7.3 – Initial bulk deployment tool" which reported the results of the work until M21. Therefore, this document may be treated as a standalone report containing design and implementation decisions of the CPSwarm Bulk Deployment Tool.

The document introduces deployment and related challenges by referring to the literature and CPSwarm technical requirements. It provides an overview of the most popular deployment solutions and evaluates their strengths and weaknesses. Furthermore, the document proposes a design aimed at providing required features while solving shortcomings of existing systems. It then presents the current system implementation, APIs, and communication models. Finally, the report concludes by summarizing the current state of the work and giving an outlook on future development opportunities.

Table of Contents

1 Introduction

Software deployment is a set of activities that make a software available for use [1]. The main deployment activities are release, installation, and activation. Release involves the steps after development cycles where a software system is assembled and prepared for transfer. Installation comprises configuring the host and the software to accommodate the execution. Activation is the process of executing the software after installation and for the first time. Other deployment activities such as update and adaptation can be considered as special forms of installation involving changes to existing software as atomic updates or adjustments respectively.

While software deployment activities can be generalized into a predefined set of steps, the nature of the steps vary from platform to platform. On top of that, a deployment on an arbitrary platform typically requiring minimal efforts can get very tedious when repeated numerous times under the same settings. As such, people involved in software deployment often rely on solutions to reduce the complexity and increase operational efficiency. It is worth noting that connected devices are growing at a fast pace such that the number of active devices is expected to reach 30.7 billion by 2020 and 75.4 by 2025 [2]. This rate has turned into a global concern for software and security experts worried about the technology readiness for such a scale. In CPSwarm Task 7.2, the consortium focuses on three main concerns within and beyond the scope of project. First, to securely roll out software updates to resource-constrained devices at large scale. Second, to ensure that software updates are delivered to devices with limited internet connectivity. Finally, to provide the ability to remotely monitor the updates and the runtime in a secure, efficient, and usable manner.

The rest of this chapter summarizes the technical requirements that are used as the basis for design and development of the CPSwarm Bulk Deployment Tool. Subsequently, Chapter 2 reviews relevant tools to identify existing solutions and possible gaps. Accordingly, Chapter 3 presents an architecture that intends to resolve existing gaps. This is followed by the implementation details in Chapter 4 which cover API design, application behavior, and an overview of the developed graphical user interface. Lastly, Chapter 5 concludes and suggests future development directions.

1.1 Requirements

The consortium has identified the need for a resource-friendly deployment tool based on the literature and over the course of numerous industrial and research projects in the past decade. While the initial analysis was reflected in the CPSwarm project proposal, an in-depth requirement elicitation was only started during the project and as part of WP2. The requirement elicitation discovered numerous pain points which are faced during daily operations by the CPSwarm application partners. These issues were gathered over the course of project and formulated into technical issues. This deliverable refers to the requirements reported until M33.

The following tables summarize the result of WP2 requirement elicitation for bulk software deployment:

[CRD-58] The Deployment Tool shall deploy artefacts on swarm members	
Description:	The generated code shall be either: • executable on the target platform • raw code with instructions on how to be compiled on target
Rationale:	This is needed to enable mass deployment on remote devices (without physical access, without exposed interfaces)
Fit Criterion:	The artefacts can be deployed to remote devices in bulks
Priority:	Minor

[CRD-59] The Deployment Agent shall report the deployment status	
Description:	The deployment status contains information about the state of the deployment, reasons for failure, and possibly log messages. Deployment Agent shall offer the possibility of reporting this information back to the Deployment Manager.
Rationale:	The status of deployment is required in order to monitor and synchronize software updates (automatically or by operators)
Fit Criterion:	The status of deployment is required in order to monitor and synchronize software updates (automatically or by operators)
Priority:	Major

[CRD-60] The communication between the Deployment Agent running on swarm members and the Deployment Manager shall be authenticated, authorized, encrypted, and integrity checked.	
Description:	• Data transmitted to and received from swarm needs to stay confidential. • Only authorized entities should be able to transmit data to the swarm members. • The confidential data received from the swarm should not be accessed by unauthorized entities. • Data received from the deployment server must be validated.
Rationale:	Secure deployments are vital for secure operation of swarms.
Fit Criterion:	All security aspects including authentication, authorization, encryption, and package signature validation are taken into account during deployment tasks.
Priority:	Major

[CRD-61] The Deployment Manager shall receive the configuration of the deployment task from the operator prior to deployment	
Description:	Deployment tool requires the configuration of the deployment which is a procedure on how (required steps) and where (target swarm members) to deploy artefacts.
Rationale:	Deployment tool requires the configuration of the deployment task to know how and where to deploy artefacts.
Fit Criterion:	Deployment Tool can be used to target specific or a group of swarm members to deploy different types of artefacts
Priority:	Major

[CRD-67] All communications between the swarm and the tools in the workbench shall be authenticated, integrity protected and encrypted.	
Description:	Deployment and monitoring should only be possible after authentication and with proper authorization. Messages in transit should be treated as confidential and must be protected against tampering and eavesdropping.

Rationale:	Updating software, setting parameters and issuing commands are sensitive operation by their very nature.
Fit Criterion:	All communications between the swarm and the tools in the workbench must use industry standard encryption and signature schemes.
Priority:	Major

[CRD-72] The Deployment Manager shall sign all packages with an operator specific key.	
Description:	The Deployment Agent should take upon itself the burden of managing the life-cycle of the main binary.
Rationale:	In order to maintain strict control over the main binary, it should only ever be started or stopped by the Deployment Agent. Before updates, it would need to be stopped anyways, and it makes signature validations before startups a lot simpler.
Fit Criterion:	The main binary should only ever be started by the designated instance of the Deployment Agent.
Priority:	Major

[CRD-73] The Deployment Tool shall implement secure over-the-air update functionality.	
Description:	This high level requirement ensures that updates can be performed securely and on CPS currently in operation.
Rationale:	One of the basic value propositions of the project.
Fit Criterion:	See related other requirements.
Priority:	Major

[CRD-75] The Deployment Agent shall verify the signatures of packages on boot and when updates are received.	
Description:	Only the Deployment Agent can verify whether the signature of the deployed packages is trusted or not, thus it should only be started by the Deployment Agent and only after the signature has been validated.
Rationale:	A signature is worthless if it is not validated each time.
Fit Criterion:	Packages (incl. binaries) with valid signatures are allowed to start. Packages with invalid signatures are not allowed to start.
Priority:	Major

[CRD-76] The Deployment Manager shall provide a way to generate, import and export operator specific keys for code signatures.	
Description:	In order to facilitate the process of signing deployment packages, the keys used need to be managed.

Rationale:	Key management enhances user experience and makes the whole process much smoother.
Fit Criterion:	The Deployment Manager can generate, import and export code signing keys and certificates.
Priority:	Major

[CRD-78] The Deployment Agent shall use the list of trusted certificates supplied when the device is first provisioned to validate signatures.	
Description:	The list of trusted CAs need to be supplied manually when the device is first provisioned - there is no way of getting around that.
Rationale:	Validating a signature is easy, but to decide whether a valid signature is trusted a list of trusted certificate authorities need to be maintained.
Fit Criterion:	Only signatures signed by a trusted CA are accepted.
Priority:	Major

[CRD-79] The Deployment Agent shall be responsible for starting, stopping and monitoring the code that has been deployed, even during startups and shutdowns.	
Description:	The Deployment Agent should take upon itself the burden of managing the life-cycle of the main binary.
Rationale:	In order to maintain strict control over the main binary, it should only ever be started or stopped by the Deployment Agent. Before updates, it would need to be stopped anyways, and it makes signature validations before startups a lot simpler.
Fit Criterion:	The main binary should only ever be started by the designated instance of the Deployment Agent.
Priority:	Major

[CRD-103] The Deployment Tool shall provide the means to compile codes on target platforms	
Description:	When compilation is required, the Deployment Tool should be able to move generated codes to target devices and compile them using provided build scripts. The build script may setup or rely on pre-existing build dependencies on the target build environment.
Rationale:	Native compilation is less complex when dealing with different hardware and software architectures on robotic systems.
Fit Criterion:	Deployment Tool offers the possibility of native compilation on target devices.
Priority:	Minor

[CRD-104] The Deployment Tool shall provide the means to cross-compile codes for the target platforms	
Description:	When compilation is required, the Deployment Tool should be able to execute build scripts that cross-compile source codes locally, before sending and installing them on the targets.
Rationale:	Cross-compilation benefits from powerful host machines and saves time when targeting similar hardware/software platforms.
Fit Criterion:	Deployment Tool offers the possibility of cross-compilation for target platforms.
Priority:	Major

[CRD-105] The Deployment Tool shall provide the means to compile codes	
Description:	When compilation is required, the Deployment Tool should be able to execute build scripts that compile codes for/on target platforms. The tool shall support cross-compilation (CRD-104) at first and then be extended to support native compilation (CRD-103) on target devices.
Rationale:	Code compilation is required when codes in compiled programming languages are being deployed.
Fit Criterion:	Deployment Tool is able to compile codes using provided build scripts
Priority:	Major

1.2 Related Documents

ID	Title	Reference	Version	Date
[RD.1]	Final Vision Scenarios and Use Case Definition	D2.2	1.0	M16
[RD.2]	Initial Requirements Report	D2.3	1.0	M6
[RD.3]	Initial Lessons Learned and Updated Requirements Report	D2.6	1.0	M14
[RD.4]	Final Lessons Learned and Requirements Report	D2.7	0.6	M26
[RD.5]	Initial Bulk Deployment Tool	D7.3	1.0	M21

2 Background Work

Previous work presents promising techniques for over the air (OTA) firmware and software update. Shavit et. al. [3] present a firmware update system which enables over-the-air update as well as diagnostics for the automotive industry. Skan [4] demonstrates a method for firmware update of flash memories on mobile devices. Other works [5, 6, 7, 8, 9, 10] offer solutions to update and monitor software running on server and cluster infrastructures. Zabbix [11] and Nagios [12] provide enterprise-class solutions for network, server, cloud, and service monitoring. While these systems provide state-of-the-art technologies in OTA software update and monitoring, they are not tailored for Internet of Things (IoT) systems which typically require high level of customization and operate with limited computing resources. These existing solutions typically target automotive industry [3], mobile devices [4], or server infrastructure [5, 6, 7, 8, 9, 10, 12, 11]. Mender [13] specifically targets IoT devices but only offers full image updates on certain platforms.

In this document, the authors review selected OTA deployment tools which could be used to address CPSwarm application requirements. Considering overall aim of the CPSwarm project to advance open solutions for a wide range of research and industry purposes, the document only analyzes tools that are available as open source.

Mender

Mender [13] is an end-to-end open-source update system for embedded Linux devices. It enabled remote secure full-image updates following a client-server architecture. Mender offers RESTful APIs to manage and monitor deployments and a UI to perform basic operations related to device managements and deployment monitoring. Figure 1 shows the architecture of Mender.

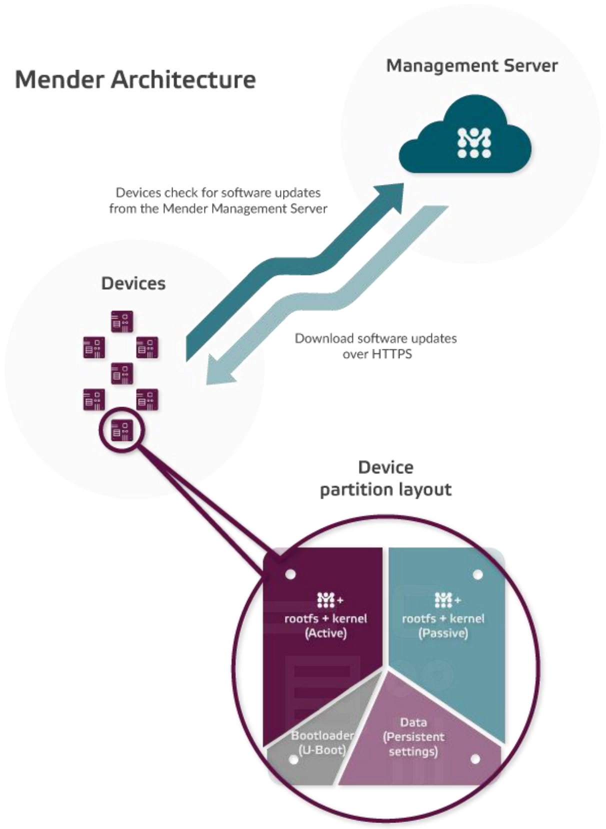

Mender Architecture
Management Server
Devices check for software updates from the Mender Management Server
Devices
Download software updates over HTTPS
Device partition layout
rootfs + kernel (Active)
rootfs + kernel (Passive)
Bootloader (U-Boot)
Data (Persistent settings)

Figure 1. High-level architecture of Mender [13].

A complete deployment using Mender involves a number of stages. First, the environment must be setup with a single Mender server and Mender clients. The clients must have a dual partition system with Mender images or a custom-built Linux image using the Yocto Project[1]. Pre-built Mender images are only available for Raspberry Pi 3[2] and BeagleBone Black[3]. Once the client devices are set and running, they authenticate with the server and continuously poll for updates. In order to perform an image update, the user should prepare a Mender Artifact which is a tarball archive consisting of the Linux image and meta fields describing the name, compatibility, type, and the image hash. The Mender Artifact can then be added to the registry using the server's RESTful API or UI and polled by target clients. The update status can be monitored at the server and in cases of failure, the clients would roll back to the previous version.

Mender provides a robust system to roll out updates to embedded Linux devices, however it only supports full-image updates and requires custom images and partition layout. Moreover, the UI does not offer any way of graphically creating or modifying the packages. It can only be used to deploy Mender Artifacts which are created in advance.

Chef

Chef [5] is an open-source configuration management tool for server applications and utilities. A Chef system consists of a Chef Server and Chef Clients where clients are typically powerful machines. The Chef Server offers a CLI for all deployment activities but does not provide a graphical UI by itself. A range of graphical features are offered by a commercial software called Chef Automate.

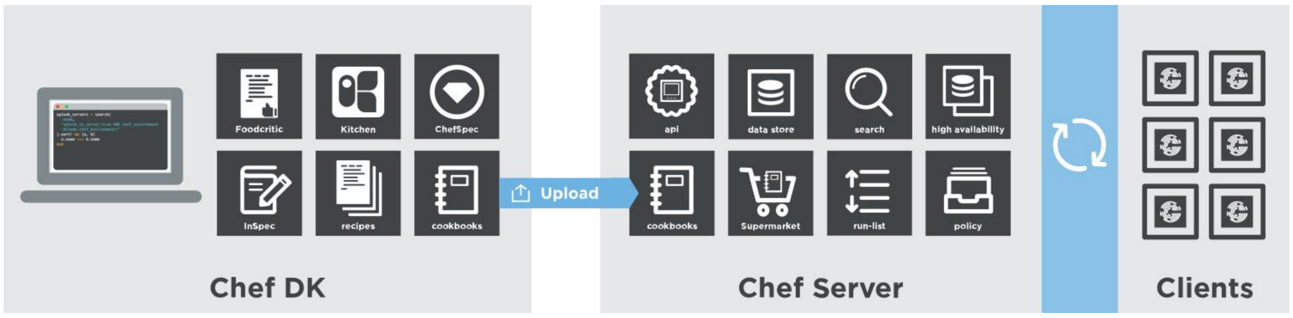


Figure 2. Architecture diagram of Chef [5].

Unlike Mender, software deployment with Chef is possible on most platform without any OS customization. The setup mostly involves the installation of Chef Server on the management server and Chef Clients on target nodes. In addition, a user requires Chef DK Workstation to interact with Chef Server for deployment operation; see Figure 2. For deploying a software, the user should write a Cookbook which uses a Ruby domain-specific language (DSL) comprising several components such as lists of files and recipes. The Cookbook itself should be placed in a Policy that also defines server type, environment, and credentials. Once a policy is submitted to the Chef Server via Chef DK, registered and authenticated Chef Clients on nodes that match the Policy will be able to fetch the Cookbook and perform the recipes. The Chef Server collects information about the status of deployments on all nodes.

Chef is a powerful system for deploying software on and configuring servers, cloud nodes, virtual machines, and network devices. Even though a Chef Client can theoretically run on CPSwarm devices, it is not tailored for environments with low resource availability. Reports [14] show that the client consumes more than 200MB of RAM during runtime which is a large amount considering the limited memory availability of CPSwarm target devices. Apart from that, there is a steep learning curve in using Chef from environment setup to a deployment. This defeats the purpose of a deployment tool that is meant to make deployments easier. Finally, the Chef Automate graphical interface is only available with a commercial license, leaving most users only with free command line interface of Chef DK.

[1] https://en.wikipedia.org/wiki/Yocto_Project
[2] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[3] https://beagleboard.org/black

Ansible

Ansible [15, 7] is another open-source tool which promises automation for cloud provisioning, configuration management, application deployment, and service orchestration. Unlike Mender and Chef, Ansible follows an agent-less architecture which leads to minimal resource usage on target environments at idle times. Ansible achieves that by directly communicating to target environments over SSH and actively executing instructions. Furthermore, Ansible can benefit from Ansible Tower [8], a commercial user interface which provides graphical configuration, deployment, and monitoring capabilities.

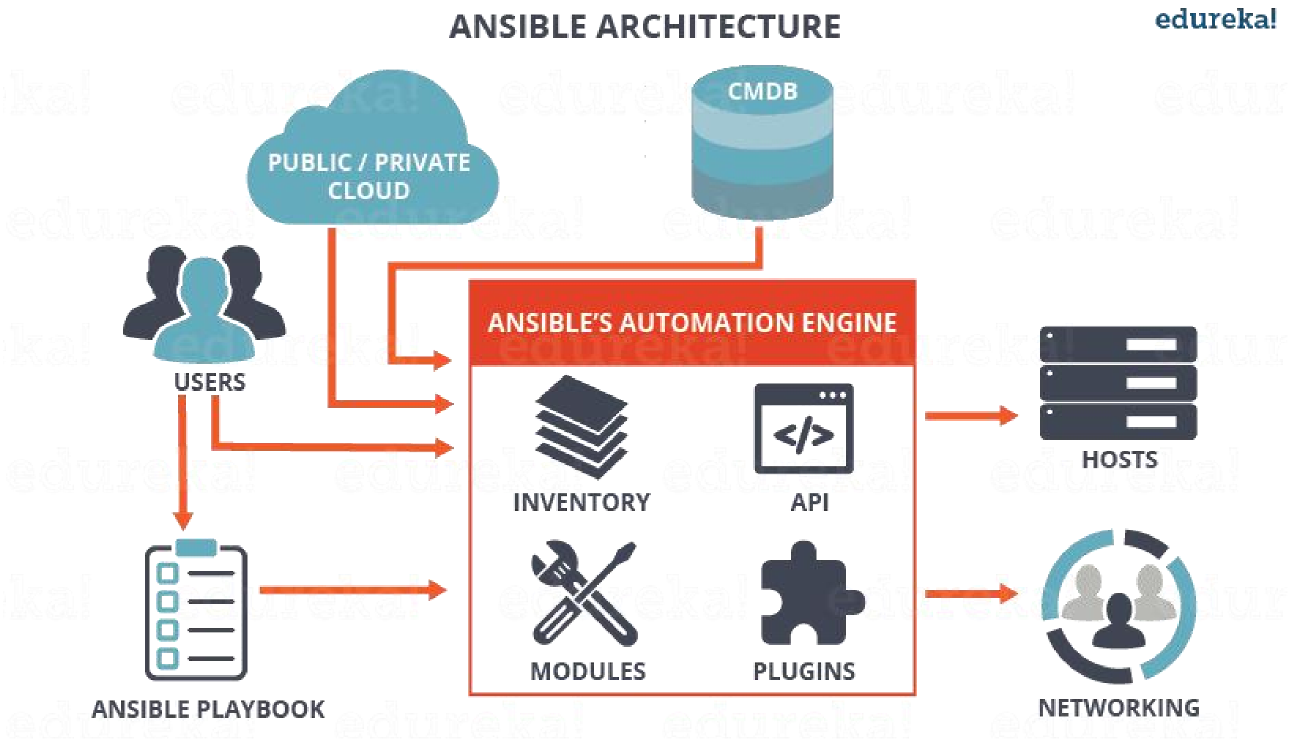


Figure 3. Architecture of Ansible [16].

In order to deploy software on target devices, the user should prepare an INI[4] inventory file and YAML[5] Playbooks. The inventory file consists of groups of devices with their hostnames or IP addresses. These devices should have active SSH servers which are accessible by the Ansible server over the provided addresses. The SSH authentication is possible using preconfigured password or SSH keys. Given the inventory file and SSH access to devices, the user will be able to execute shell commands remotely on all or particular groups of devices. Alternatively, the user can write a Playbook which orchestrates the operations that should be executed on the groups of devices. Figure 3 illustrates the architecture of Ansible.

Ansible provides a simple and efficient toolset for configuration and deployment automation of different hosts. However, it is not suitable for IoT systems due to a number of architectural issues. First of all, Ansible relies heavily on SSH and benefits from its ubiquity in major operating systems. Although SSH servers may exist on target platforms, it does not mean that they are always accessible to the Ansible server. A normal connection through SSH requires an active networking (TCP/IP) link and access to the SSH server's bind port over a public IP address. This is not the case for most IoT systems deployed in the field and possibly using cellular or limited networks. If feasible, NAT port forwarding[6] may overcome this issue but adds to the complexity. Moreover,

[4] https://en.wikipedia.org/wiki/INI_file
[5] https://en.wikipedia.org/wiki/YAML
[6] https://en.wikipedia.org/wiki/Port_forwarding

relying on SSH means that Ansible server requires the current public IP address of devices or a domain name that translates to the correct IP address by an external DNS. IoT devices are volatile networking nodes and often communicate with dynamic IP addresses. Secondly, Ansible is an agent-less system and consumes target platform resources only during a deployment activity. This saves a lot of computing resources that would otherwise be used by an agent or client, however it leads to a lack of host environment awareness and may negatively affect the system during deployments with the risk of exhausting primary or secondary storages. Finally, the Ansible Tower UI which is provided for graphical deployments and monitoring is a commercial product. As a result, users who want to graphically deploy on and monitor a large number of devices must pay a very high subscription fee.

Salt

Salt or SaltStack [9] is an open source project for configuration management, remote execution, and event-driven provisioning. Salt provides extra flexibility by allowing both agent-based and agent-less operations. The enterprise version of Salt offers a GUI with features for monitoring target systems [17].

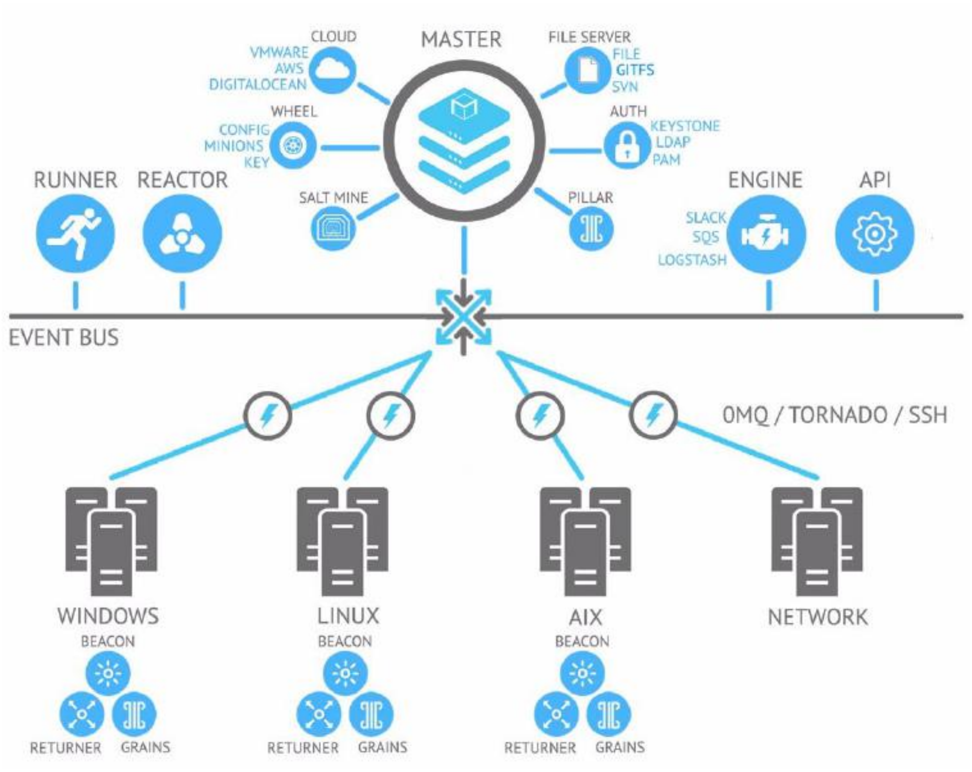


Figure 4. Architecture of Salt [10].

To deploy software on multiple devices using default settings, one must setup Salt Master on the server and Salt Minions on target devices; see Figure 4. The Minions find the Master using the default address or a parameter given in the configuration file. Minions and Master authenticate using public-key encryption and authentication. Each Minion needs an ID which is pre-configured or generated using device's fully qualified domain name (FQDN) or hostname. It also requires the public key of Master in place. When started, the Minion creates a client key-pair and submits its public key as to the Master for authentication. These authentication requests can be managed using a CLI interface of Master. Each Minion keeps device information such as operating system and CPU architecture in Grains. This information is kept in Master's Pillar and can be used along with other labels and IPs to target devices. Salt allows execution of single commands or States on targeted devices. A State is a YAML document with different sections describing every required configuration on the targets. These include package dependencies, file structure, required services, and files that should be copied from master to targets. Multiple States can be placed together in a Top file for bulk configurations.

Single commands, States, and Top files can be submitted via the CLI, a Python SDK, or an HTTP API reporting results in a textual structured format.

Salt is a powerful system with a rather flat learning curve. It enables bulk deployments using a human-readable YAML specification with a wide range of high-level utility functions to help perform common operations. Similar to Chef [5] and Ansible [7], Salt is designed for remote configuration of sever infrastructures. As a result, Salt focuses on providing server configuration capabilities without worrying so much about runtime footprints. The agent-less version of Salt facilitates zero-resource consumption idle times by relying on an existing SSH server. However, this does not guarantee low resource consumption during operating times and works only when there is a possibility of opening SSH connections to targets. With the Master-Minions version of Salt, all Minions connect to the centralized Master which publishes notifications whenever there is an update. Compatible Minions then make independent requests to the Master asking for the update package. This form of update distribution results in inefficient network usage where a single package must be transferred from the server as many times as clients are. In addition, the notification followed by concurrent request-replies may cause congestion and negatively affect the whole network.

The CPSwarm Bulk Deployment Tool provides similar features but focuses on those that are most relevant to IoT scenarios. It will reduce the learning curve by providing simpler interfaces that are intuitive for a wide range of users. The communication, choice of protocols, and operations will address shortcomings of existing systems with respect to the domain requirements.

3 Architecture

The authors analyzed the CPSwarm project requirements and weaknesses of existing deployment systems to design a software tailored to project needs. These requirements along with the project use-cases [RD.1] serve as the basis for the design and evaluation of this deployment system. With these considerations, the authors have formulated four essential factors in design of the CPSwarm Bulk Deployment Tool:

- The proposed system should reduce the complexity of bulk, over-the-air deployment of software by providing simple interfaces and a flat learning curve. (Usability)
- All components of the system, especially those that operate at the edge, shall run with minimal footprints during both idle and operating times. (Efficiency)
- The system should offer features that are required for over-the-air bulk deployment of software on IoT devices. (Practicality)
- The system should follow state-of-the-art practices to ensure security during all deployment operations. (Security)

These factors have taken a principal role in all iterations of design, implementation, and analysis over the course of the project.

This chapter presents the final system design by providing an overview of terminology, security considerations, and software components. The rest of the document often refers to CPSwarm Bulk Deployment Tool as CPSwarm Deployment Tool or simply the deployment tool.

3.1 Concepts and Terminology

Considering usability as one of the key factors in the design of the Deployment Tool, the authors emphasize on minimizing the introduction of new terms and instead use familiar terms in software deployment domain.

The terms are defined below:

Target
Target is a physical device with an operating system and update capabilities. Each target is identified with a unique ID and a set of tags (e.g. device type, group).

Build
Build refers to compiling, minimizing, and packaging a software for re-use on other devices. The build may be performed on any host with necessary computing resources. Such a host may be a device which has a strong power source, good hardware, or appropriate build dependencies. Alternatively, the host may be a Docker container (on the same or another server) with appropriate cross-compilation toolchain. The build should result in artifacts that can be transferred and installed on other devices. It is also possible to install on the same device that was used for the build.

Installation
Installation is considered as placing an archive into the right place and preparing it for execution. The installation may include building the package or installation of dependencies.

Task
Task is a set of instructions and configurations which describe an intended deployment process. This process includes typical deployment steps such as build, installation, and package runtime. In addition, the Order provides information about target devices and logging requirements. The tasks can be categorized into two types depending on the configuration. Tasks with build steps result in Build Tasks and those with installation and runtime commands results in Deployment Tasks.

The following concepts explained here for better understanding of the architecture:

Target Selection
The Deployment Tool creates an abstraction on top of the transport layer and enables device identification using IDs and tags. As such, the users do not need to worry about IP addresses in rather dynamic IoT settings. During a deployment, selection of few devices is simple enough using their unique IDs. However, as the number of devices grow, a logical grouping becomes necessary to ease bulk deployments. The deployment tool uses tags for grouping of devices. Tags can be used in two different ways:

- Set during the device configuration to identify the target based on static specifications. These include hardware architecture, operating system, and device type. (e.g. arm, linux, drone)
- Set during the device configuration or added during the operation to identify the target based on dynamic information. These tags include name of the location where the device operates in and labels describing current responsibility of the device. (e.g. digisky, hanger, crewA)

The user can use these tags for perform bulk deployments without the need to know or specify the individual device IDs. The system performs the deployment on the devices that match either the ID or tag. This document refers to these devices as *matching targets*.

Target Discovery
In order to perform a deployment, both user and server should be aware of the available targets. One way to achieve this is by manually adding each target to the server. However, this would get very tedious when the number of devices is large. Instead, the Deployment Tool relies of target advertisements published by the targets themselves. This is coupled with a secure registration flow which ensures the registration of trusted targets only.

Secure Deployment
The Deployment Tool ensures the secure deployment of software components by multiple measures:

- The system only allows deployment of software on target devices by operators who have the specific authorization. This is achieved by means of access control at the API level on the server side.
- The registration of target devices to the server is authenticated and authorized. This is to make sure that only devices that are known and configured by the operators get registered into the server and be able to initiate a connection.
- The communication link used to transmit the software package is encrypted and authenticated. Different parties exchange their public keys during the registration phase. These keys are then used to authenticate connection requests and encrypt the traffic in both directions. This protects the integrity and confidentiality of deployed packages and logs during the deployment.
- The system prevents tampering of deployed packages on devices by operating in privileged user space. This offers the same level of integrity to packages as to the key pair used for communication. This protects the packages from users and processes with unprivileged access but does not prevent privileged or physical tampering.

3.2 Components

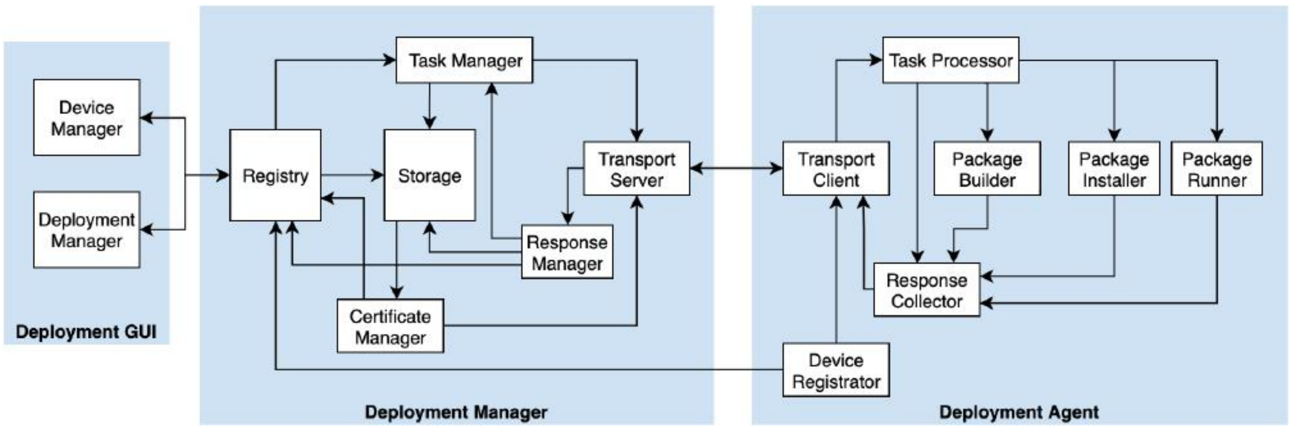


Figure 5. Conceptual diagram of the CPSwarm Deployment Tool

The CPSwarm Deployment Tool follows a client-server architecture with lightweight client components tailored for resource constrained environments, a scalable server component, and a graphical user interface to enhance the system usability. All components follow a modular design with low coupling and high cohesion; see Figure 5. This enables iterative development and maintenance of the system in a simple and structured manner. The server-side component, called Deployment Manager, is a centralized component with interfaces for user interaction and client communication. On the other hand, the client-side component, called Deployment Agent, runs on every device with very low footprints. The Deployment GUI is a web application that offers a graphical interface for the Deployment Manager and for simplifying various deployment operations. This section briefly introduces these components. The next chapter provides a more technical description of the implemented components.

3.2.1 Deployment Manager

The Deployment Manager is the central component of the Deployment Tool with APIs to communicate with other components. Even though this is a centralized component, it is able to handle large amounts of traffic by vertical scaling. The Deployment Manager is developed with operation concurrency in mind in such a way that available resources are utilized efficiently during high load. Furthermore, the manager makes intensive use of queueing mechanisms to process high load of requests without causing congestion. This form of vertical scaling enables management of up to hundreds of targets. The system can further scale horizontally by instantiating multiple managers and load balancing at the API level. Technical guidelines on horizontal scaling of the CPSwarm Deployment Tool is beyond the scope of this document.

The Deployment Manager sub-components are described below:

Registry
The Registry is the set of APIs and controllers for processing target and deployment information. It exposes APIs which allow management of targets and tasks. It is also possible to subscribe to notifications to get live data such as status of targets and task logs. The registry maintains all the information in catalogues that are accessible in a RESTful style. The API definition is provided in Chapter 4. The Deployment GUI makes extensive use of these APIs. Moreover, the Deployment Agent uses the APIs to register new target devices.

Storage
The Storage is a key-value store to persist different kinds of information. This component interfaces with external database systems to provide high performant storage and retrieval of data. The selected database system should scale according to the runtime requirements. Additionally, the database should support data

encryption on the secondary storage to protect the deployment data. The storage component interfaces with database system through connectors.

Task Manager
The Task Manager is an internal component which processes the information from the Registry and prepares tasks for transfer to target devices. The status of the preparation and transfer of each task is sent to the Storage component for persistency. For tasks that involve both build and deployment, the task manager sends the build to a designated host, waits for completion acknowledged by log manager, and then sends the resulting build artifacts to target devices. Each task is sent in two stages. The first stage is an announcement, containing task identity and artifacts size. This information is used by the target device to assess whether it can accommodate this task without exhausting local resources. The devices which approve, subscribe to the second stage which contains the actual artefact bytes and instructions for the deployment.

Certificate Manager
An internal component that provides utility functions related to certificate generation and key management to be used by other components for authentication and encryption. On one hand, it supports the Registry for validation of keys during device registration. On the other hand, it loads existing keys from Storage and inserts them into the Transport Server for authentication requests from clients as well as for data encryption and decryption.
The Certificate Manager also takes part in the certificate-based admission control (the process of joining the swarm) feature of the CPSwarm Communication Library by acting as the trusted component of the system, who issues a certificate (a signed public key - the public part of uniquely generated public-private keypair for every swarm member).

Transport Server
The component enables secure, reliable, and efficient message exchange with the Deployment Agent over the network. It utilizes appropriate encryption, compression, and chunking techniques depending on the protocol and requirements. Furthermore, it applies message queuing and load balancing to manage large amounts of traffic without overloading other components of the system. The system may offer multiple implementations of the transport server utilizing different protocols, addressing specific use-cases. Every implementation should provide all the required functionalities.

Log Manager
This component processes messages that are received by the Transport Server from the Deployment Agent. Overall, the log manager deals with three kinds of messages:

- The meta information about target devices carrying live status of a device such as its tags and location.
- The status of the tasks including process logs and deployment progress
- The result of build tasks containing compressed software artefacts

The component processes and pipes the information to Storage or the Task Manager. Furthermore, it sends events data to Registry for live subscriptions by the Deployment GUI.

3.2.2 Deployment Agent

The client-side component of the CPSwarm Deployment Tool that runs on every target device. The design and implementation of the Deployment Agent place maximum focus on reducing runtime footprints. This is to ensure that the limited resources available on devices are kept available for other running application to the greatest degree. The Deployment Agent is mostly responsible for receiving tasks from the manager, validating and installing them, and afterwards managing their runtime lifecycle. A Deployment Agent could also be used to perform builds for other devices. Logging considerations at every step of the deployment assist developers in discovering deployment issues and adds transparency to remote devices.

The sub-components of the Deployment Agent are described below:

Transport Client
This component connects to the Deployment Manager's Transport Server. It receives and processes messages depending on the protocol, messaging pattern, and other requirements. This component takes care of communication security, reliability, and efficiency, similar to transport server but on the client side. It also controls the incoming traffic by means of message queuing to prevent over-loading of the Task Processor.

Task Processor
The Task Processor is a controller in charge of high-level operations and orchestration of other activities. It receives messages from the Transport Client. It evaluates task announcements for compatibility with the hosting target and feasibility of processing and installing them without exhausting the system resources. If a task is acceptable, it will issue a subscription request to the Transport Client. Once the actual task is received, the processor decompresses and writes any enclosed artifacts to a temporary location. It will then send instructions to other components such as Package Installer and Package Runner. Build instructions are forwarded to Package Builder. Furthermore. the task processor handles a range of other requests such as log requests, terminal command executions, and stop signals. These requests are described in Chapter 4.

Package Installer
The Package Installer performs all installation steps of a deployment task. This includes executing provided commands, and if needed, removing previous deployment files. Package Installer follows installation steps sequentially and aborts the installation in case of failure in a step. A successful installation may be followed by removal of files that are no longer needed on the target. Failed or successful installation status information is reported to Response Collector. More verbose installation logs are sent to Response Collector and kept in a buffer for debugging purposes.

Package Runner
Some deployment tasks include one or more runtime steps. The Package Runner executes these commands as sub-processes and manages their lifecycle throughout the Deployment Agent runtime. A successful or erroneous termination of the sub-processes along with verbose logs are reported to Response Collector. Interrupted sub-processes are re-created when the Deployment Agent is restarted.

Package Builder
The Package Builder is a special wrapper for Package Installer intended for building a package for other devices. This component enables the use of a device with appropriate dependencies and processing capabilities as a platform for various compilation needs. Similar to the Package Installer, the builder executes the given commands sequentially and aborts in cases of errors. Moreover, the builder compresses the expected artifacts and sends them to Response Collector for transport to the Deployment Manager. The build progress and verbose build logs are also sent to the Response Collector.

Response Collector
This component collects status and log messages from other components, serializes and sends them to Transport Client. The status and error logs are collected and sent in batches every few seconds. Other verbose build, installation, and runtime logs are kept in a limited size buffers and sent only in debug mode or when requested explicitly by a user from the server.

Device Registrator
The Device Registrator is responsible for registering the device to the server for the first time. When setting up the Deployment Agent, the user provides a token that is generated by the server. The registrator generates a key pair and submits the public part along with device identity information to the server. The token is a part of this request to authenticate the device to the server. The registration process is described in detail in the implementation chapter.

Moreover, the Device Registrator supports the CPSwarm Communication Library by generating a secondary key pair. During the device registration, the public part of the key pair is sent the Deployment Manager. The Certificate Manager signs the public key to form a certificate and sends it back to the device along with the public key of the server, which can be used for certificate validation.

3.2.3 Deployment GUI

The Deployment GUI is a web application that offers a graphical user interface for supporting various deployment operations. The user interface tries to tackle usability aspects of bulk deployment by introducing novel interaction methods in the whole deployment workflow. These involve intuitive ways to configure devices, roll out deployments, monitor the status and progress, and to diagnose issues more effectively.

The Deployment GUI consists of two sub-components described as follows:

Device Manager
The Device Manager includes utilities which are directly related to individual devices. Users are able to view devices, modify meta information such as tags and location, or remove devices. Discovered devices are those that connect to the server for the first time, carrying a valid token issued previously via the UI. Deleting a device is equivalent to revoking its access. Each device has a timeline showing the history and status of past deployments. A terminal provides Shell-like access to the device for command execution and hands-on debugging. Device location is optional and set manually by the user or produced by external positioning sources on the device. Devices with location are displayed on a map.

Deployment Manager (GUI)
The Deployment Manager consists of core UI capabilities supporting various deployment activities. In case of package builds, the user is enabled to select source files, provide a set of commands for the build, specify artifacts that need to be exported after the build, and finally the host device that performs the build. The host is any device, virtual machine, or container that runs an instance of the Deployment Manager and has resources to build the package. The artifacts of the build are collected and maintained on the server. These artifacts could be submitted to, installed, and ran on one or more targets. For installing packages and running the applications, the user can provide specific executable commands. Furthermore, the user should select target devices by searching names, tags, or by picking them on the map. Working with tag and the map enables bulk device selection. After triggering the deployment, the user will be presented with progress information and an interface to request and view device logs.

4 Implementation

This chapter present the implementation of CPSwarm Bulk Deployment Tool based on the final design. The current implementation addresses all functional requirements which make the tool usable in laboratory environments. The implemented security mechanisms enable the use of this tool in public networks, maintaining the authority, integrity, and confidentiality of the software.

The CPSwarm Deployment Tool is released with an Apache 2.0 license and publicly available on CPSwarm's Github organization [18]. Interested parties may access the complete project source code, documentation, and released binary distributions for using the Deployment Tool in a range of cyber physical applications. The authors encourage feedback, issue reporting, and external contributions.

The implementation is on the basis of the four design factors (usability, efficiency, practicality, security), described in the Chapter 3. The system offers utilities that are practical for the target domain and avoids turning into a feature creep. This helps in maintaining the simplicity of the system and at the same time eases future developments. The offered functionalities are secure by design with mandatory enforcements ensuring different security aspects. The mandatory security adds minimum overhead in terms of system usability and communication. Moreover, the components of the system are implemented to run with low footprints and suitable for constrained environments. Software deployment is required very seldomly, thus the toolset to support it should not have long term computational needs. Lastly, the system offers a graphical user interface with the intention of enhancing usability and adding full visibility to bulk deployment activities.

The system is developed in Go programming language, an open source compiled language with memory safety, garbage collection, and CPS-style concurrency [19]. The features and the strong built-in libraries of Go make it an ideal language for developing a reliable and efficient program with high concurrency and a simple code base.

The storage backend of Deployment Manager uses Elasticsearch[7] which provides a high performant API for storage and retrieval of data. Elasticsearch can be deployed as a centralized node or distributed. In either form, it is possible to scale storage, queries, data ingestion as well as backup and recovery.

The following sections describe the implementation starting from external interfaces, diving into the logic of internal components, and a brief overview of the graphical user interface.

4.1 External Interfaces

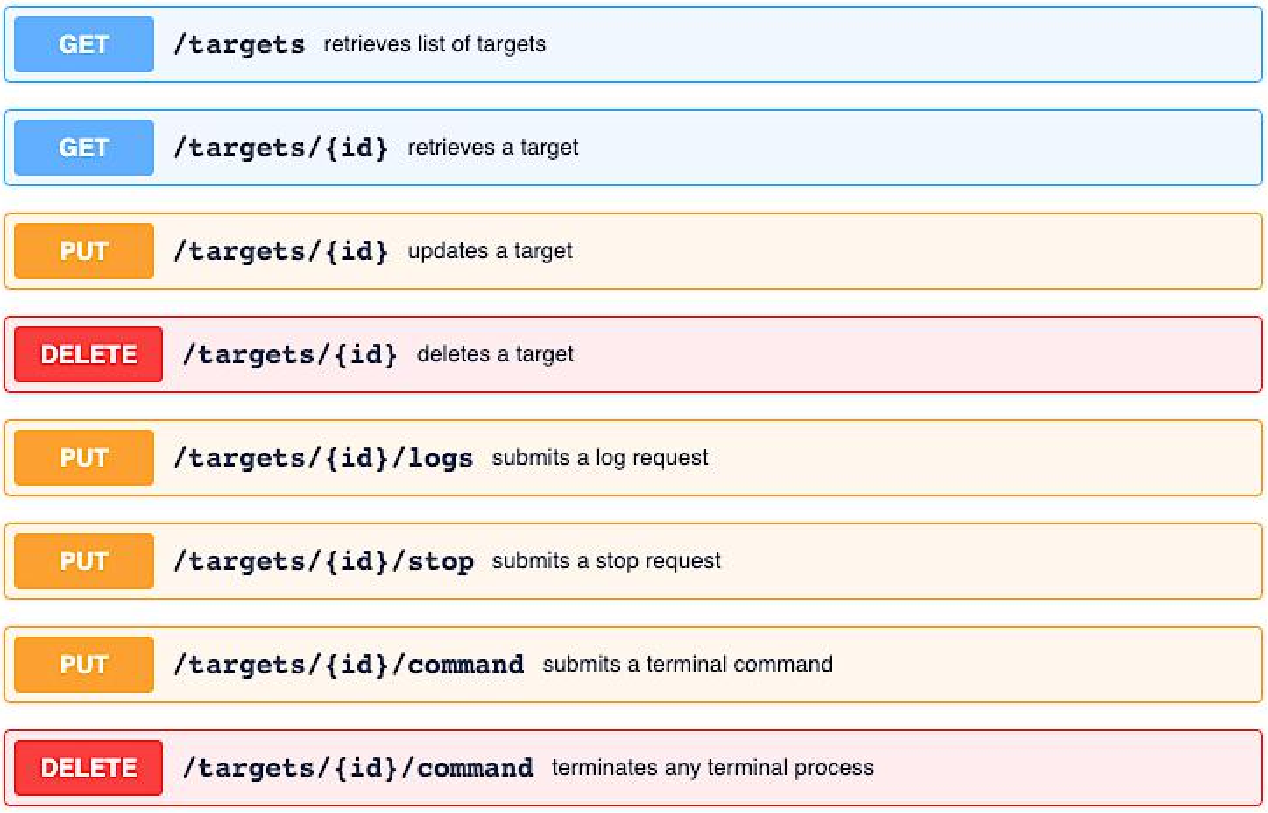
These external interfaces are exposed over the network, consumed by users and graphical interfaces. All APIs support JSON[8] as serialization formats. Some APIs additionally support YAML[9] for easy hands on testing. While JSON is highly portable and readable, YAML provides better writability when configuring tasks by hand. The API specifications are described in Open API 3.0[10] format and are available as part of the documentation. The Open API specifications include the operation and data model and may be used to generate server/client stubs as well as human readable documentations. The implemented APIs are described as follows.

Targets

The Targets API is a RESTful API offering endpoints to fetch the list of targets, read, update, and delete individual ones. Apart from these, several endpoints allow making specific requests to targets. These requests

[7] https://www.elastic.co/products/elastic-stack
[8] https://en.wikipedia.org/wiki/JSON
[9] https://en.wikipedia.org/wiki/YAML
[10] https://en.wikipedia.org/wiki/OpenAPI_Specification

are asynchronous and return as soon the request is sent to the target. In case of log requests, the response arrives at a later stage which can be received by subscription on the Events API or by making a query to Tasks API. There is an endpoint for requesting a termination of all active tasks and their associated processes on the target. Furthermore, two endpoints allow managing single commands to support remote command executions on a target. Figure 6 lists the endpoints.



GET `/targets` retrieves list of targets

GET `/targets/{id}` retrieves a target

PUT `/targets/{id}` updates a target

DELETE `/targets/{id}` deletes a target

PUT `/targets/{id}/logs` submits a log request

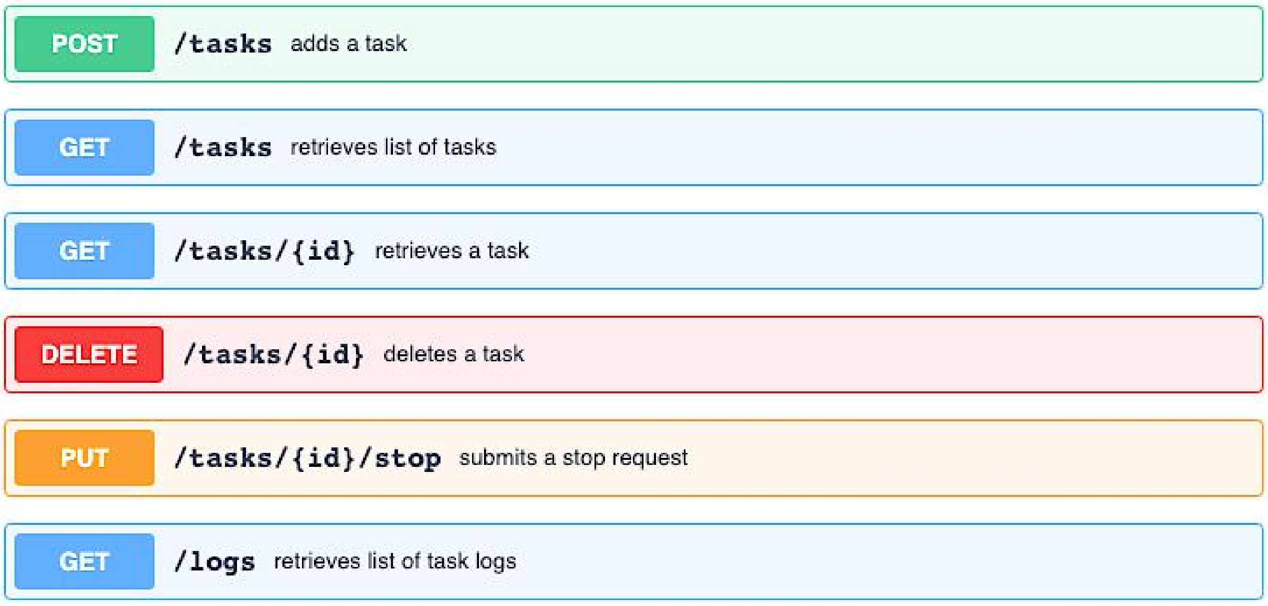
PUT `/targets/{id}/stop` submits a stop request

PUT `/targets/{id}/command` submits a terminal command

DELETE `/targets/{id}/command` terminates any terminal process

Figure 6. The list of endpoints in Targets API, generated from the Open API specifications.

Tasks
The RESTful Tasks API provides endpoints to list all tasks, create, read, and delete them. The API does not offer a way to update tasks since a submitted task immediately starts the deployment. Possible modifications to a deployment should be submitted as a new task building on top of the previous task or replacing it in case of idempotent deployments. Deleting a task is only possible if there are no running processes. A separate endpoint provides a way to terminate running processes of a task. Finally, an endpoint offers comprehensive querying possibilities of task logs. This endpoint accepts query parameters to filter logs based on target, task, deployment stage, executed command, and a full text search of log messages powered by Elasticsearch's engine. Moreover, the endpoint allows filtering error logs and sorting of logs by parameters such as time, target, and task. The log requests are responded in paginated form to reduce the load on both server and client. The list of endpoints is shown in Figure 7.


POST `/tasks` adds a task

GET `/tasks` retrieves list of tasks

GET `/tasks/{id}` retrieves a task

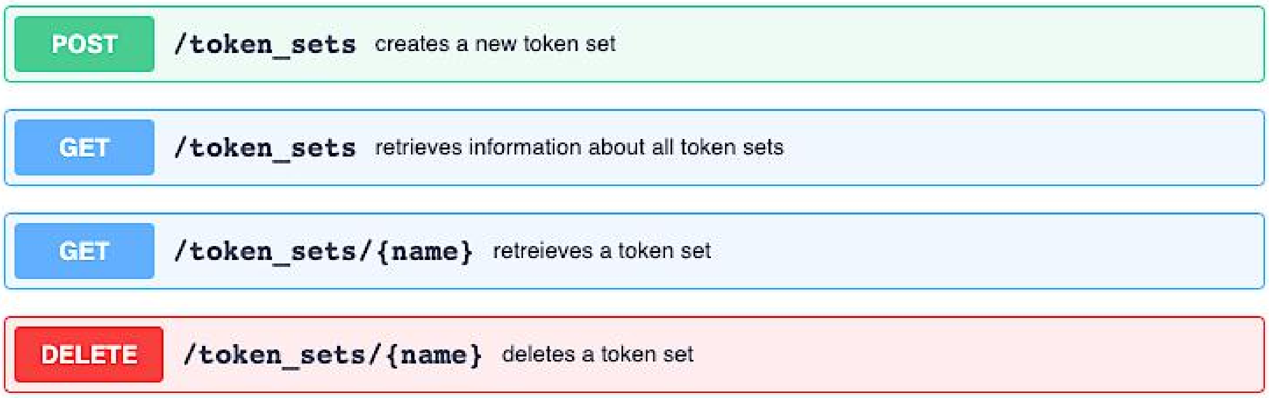
DELETE `/tasks/{id}` deletes a task

PUT `/tasks/{id}/stop` submits a stop request

GET `/logs` retrieves list of task logs

Figure 7. The list of endpoints in Tasks API, generated from the Open API specifications.

Tokens
The RESTful Tokens API provides endpoints for token management as part of the target registration process. Tokens are created in groups to associate them with individual deployment setups. Each group requires a name and may contain minimum one token. When requesting for tokens, the user provides the group name and the total number of required tokens. The server generates the required number of tokens and stores the hashed version in the database. The user gets these tokens in response along with an expiry date for the set. The user may later on query the expiry date or delete the set, but it is not possible to query the tokens as the server only maintains the hashed version. The list of endpoints is given in Figure 8.


POST /token_sets creates a new token set

GET /token_sets retrieves information about all token sets

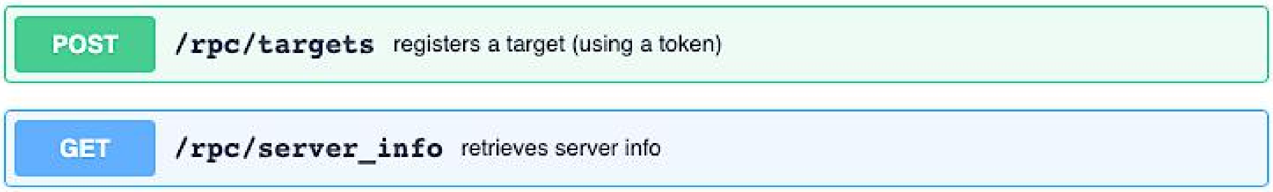
GET /token_sets/{name} retreieves a token set

DELETE /token_sets/{name} deletes a token set

Figure 8. The list of endpoints in Tokens API, generated from the Open API specifications.

RPC

The RPC API are service endpoints which are exposed externally but are meant for use between the Deployment Manager and Deployment Agents. One endpoint is used for target registration using a token obtained from the Tokens API. The Deployment Agents submits the target info including the ID, tags, location, and pubic key and if the token is valid, the server registers the device. Another endpoint is used for querying public information about the server such as the public key and protected communication endpoints.


POST /rpc/targets registers a target (using a token)

GET /rpc/server_info retrieves server info

Figure 9. The list of endpoints in RPC API, generated from the Open API specifications.

Events

In addition to the RESTful HTTP APIs, the system exposes a notification channel based on the WebSocket[11] protocol. This channel can be used by client applications such as GUIs to get the latest state of targets as soon as this information becomes available to the manager. Subscribing to events are preferred over short polling of other APIs.

The Websocket is exposed at /events endpoint. The client subscribes to all events, except when the list of events is passed in topics query parameter. The following events are available:

- logs (plural) - additionally supporting target and task query parameters
- targetAdded
- targetUpdated

[11] https://en.wikipedia.org/wiki/WebSocket

4.2 Internal Interfaces

Internal interfaces are exposed over the network and are only used between the Deployment Manager and Deployment Agent. An analysis by the consortium compared protocols such as XMPP[12], MQTT[13], DDS[14], and ZeroMQ[15] based on a number of factors. This analysis is provided as Annex A - Security Workshop - March 26-27th 2018, Budapest. As a result, ZeroMQ was selected for all the communications to target devices.

The deployment tool uses ZeroMQ for every communication between the manager and the agents. The authors make intensive use of the publish-subscribe pattern enabling communication over TCP as well as Pragmatic General Multicast (PGM)[16]. PGM provides reliable multicast mechanism over UDP, enabling efficient use of the network for many of the manager-to-agent communications. If needed, the modular design of the system allows addition of support for other publish-subscribe protocols such as MQTT.

The messages exchanged between manager and agents are serialized for better portability over the network. The current version of the deployment tool uses JSON as the serialization format because of its simplicity and human-readability during ongoing development stages. A compact JSON serialization adds negligible processing time and message size [20]. Future work may consider utilizing the Protobuf[17] protocol to further reduce the serialization bottleneck. This may reduce message sizes by around 30% and processing times by 80% on manager and agents [21].

Deployment Manager exposes publisher and subscriber interfaces for the following:

Advertisement and Discovery
The advertisement and discovery mechanism make it possible for Deployment Agents to advertise their existence, identity, and status to the manager. This information is needed to maintain a registry of targets as well as to establish secure communication link for ZeroMQ. The advertisements are published to the manager.

Task Announcements
A task announcement is published to all matching targets as soon as a deployment task is ready for transfer. The announcement includes the task ID and archive size. The agents running on targets receive the announcement and assess the possibility of processing the task given its size and available system resources. The agents send the result of the assessment to the manager. If processing the task is possible, the agents subscribe to the task topic waiting for the actual task.

Tasks
The task includes the compressed package, installation, and runtime instructions. It is published to all matching targets which have assessed the announcement and subscribed to the task. Once an agent receives the task, it unsubscribes from the topic.

Acknowledgements
Small messages published to the manager for status reporting. The acknowledgement consists of the target ID, task ID, and the status code. These messages are sent at different stages of the deployment to inform the manager about the progress.

[12] https://en.wikipedia.org/wiki/XMPP
[13] https://en.wikipedia.org/wiki/MQTT
[14] https://en.wikipedia.org/wiki/Data_Distribution_Service
[15] https://en.wikipedia.org/wiki/ZeroMQ
[16] https://en.wikipedia.org/wiki/Pragmatic_General_Multicast
[17] https://en.wikipedia.org/wiki/Protocol_Buffers

Logs

Logs are extended acknowledgement messages including details about an event. Events are errors from the agent or standard output/error of executed commands. Depending on the deployment configuration, events are published as soon as they happen or only when requested. The request for logs is sent via the manager's Targets API.

4.3 Application Behavior

This section illustrates the process behind a simple deployment to help in understanding the use of Deployment Tool APIs in practice.

Figure 10 shows the sequence of operations in a single deployment (install and run) on one target device.

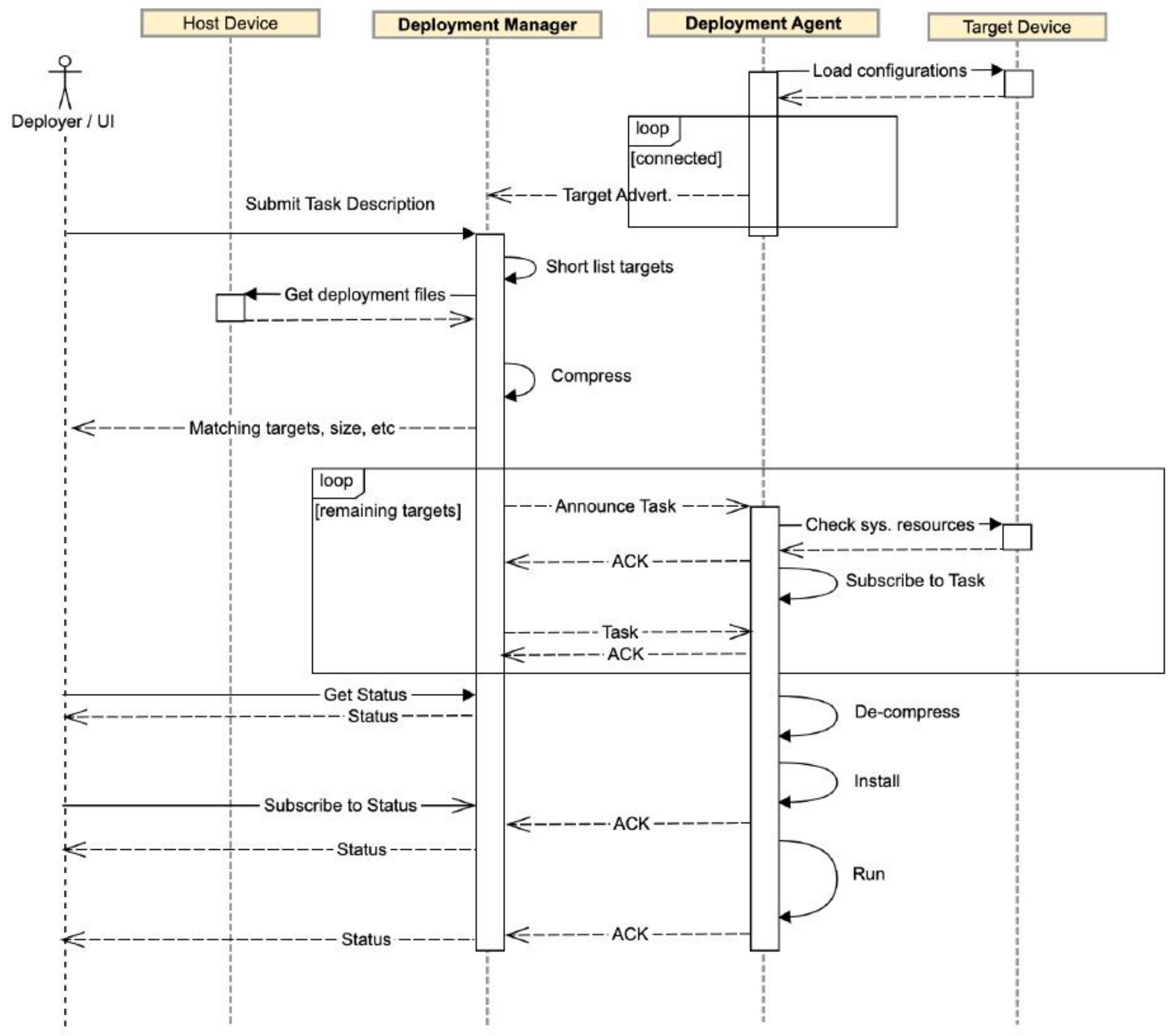


Figure 10. Sequence diagram of deployment on a single target.

The diagram shows the deployment assuming the target already registered into the system. This registration process happens prior to any deployment and involves an operator which has access to both the server and the device. As explained before, the device and server talk over two protocols: ZeroMQ and HTTP. ZeroMQ is for publish-subscribe communication (sending software updates, getting back status and logs) and HTTP is for management tasks over the RESTful API. The Deployment GUI makes extensive use of the RESTful API.

The secure communication between Deployment Agent and Deployment Client is implemented using the CurveZMQ[18] protocol. This protocol follows the CurveCP security handshake [22] and is built to add security to ZeroMQ. The protocol intends to provide secure messaging by preventing "eavesdropping, fraudulent data, altered data, replay attacks, amplification attacks, man-in-the-middle attacks, key theft attacks, identity attacks, and certain denial-of-service attacks" [23]. Internally, this protocol uses Curve25519 elliptic curve [24] which "establishes short-term session keys for every connection to achieve perfect forward security. Session keys are held in memory and destroyed when the connection is closed. CurveZMQ also addresses replay attacks, amplification attacks, MITM attacks, key thefts, client identification, and various denial-of-service attacks" [23]. However, the protocol does not specify a mechanism for key exchange, leaving this to the individual applications. The Deployment Tool handles the key exchange in a secure way tied with the device registration. The design puts emphasis not only on security but also on the usability. Figure 11 shows the sequence of operations for the registration. The parts of the process marked with red adds the steps required for creating the certificate for the CPSwarm Communication Library.

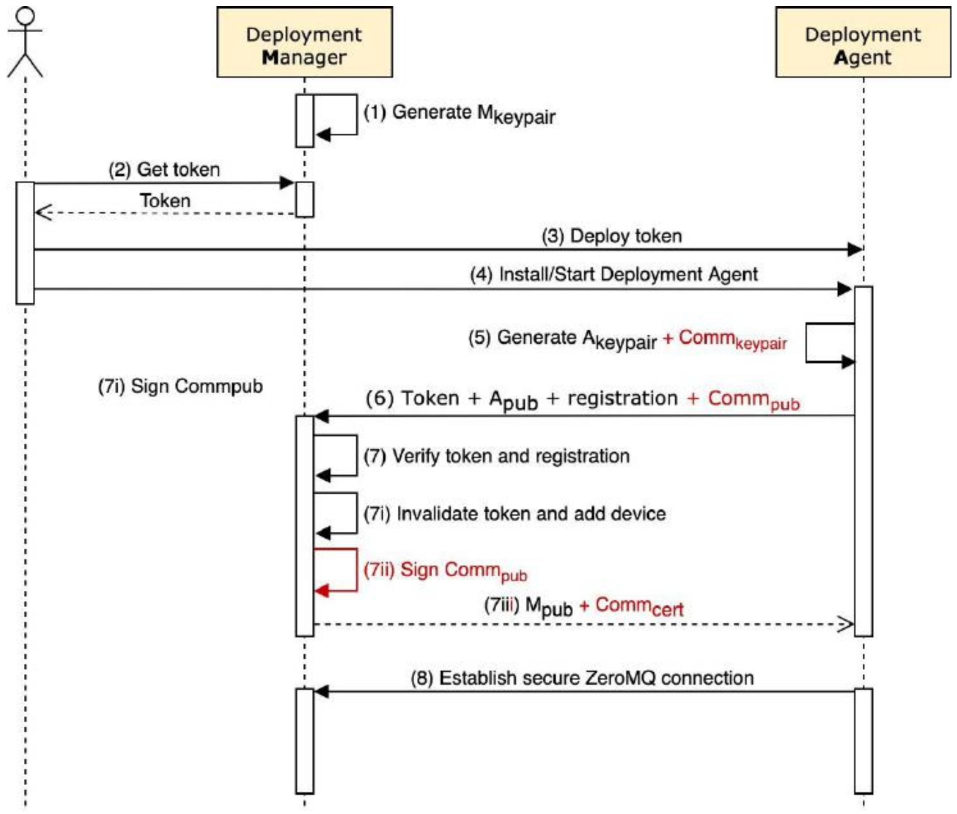

Deployment Manager
Deployment Agent
(1) Generate $M_{keypair}$
(2) Get token
Token
(3) Deploy token
(4) Install/Start Deployment Agent
(5) Generate $A_{keypair}$ + $Comm_{keypair}$
(7i) Sign Commpub
(6) Token + A_{pub} + registration + $Comm_{pub}$
(7) Verify token and registration
(7i) Invalidate token and add device
(7ii) Sign $Comm_{pub}$
(7iii) M_{pub} + $Comm_{cert}$
(8) Establish secure ZeroMQ connection

Figure 11. Sequence diagram of the secure target registration flow.

The registration flow works as follows:

1. The Deployment Manager generates a persistent CurveZMQ key pair ($M_{keypair}$).
2. A user authenticates and asks for a token from server over the RESTful API (TLS/CA-signed certificate). The authentication is based on the OpenID Connect[19] standard. The retrieved token is an ephemeral key (read below).
3. The user manually copies the token to device.
4. The user installs the Deployment Agent on device, which starts a background service.

[18] http://curvezmq.org/

[19] https://openid.net/connect/

5. The Deployment Agent generates a persistent CurveZMQ key pair ($A_{keypair}$) and the persistent communication key pair ($Comm_{keypair}$). The communication key pair generation is provided by libsodium[20].
6. The Deployment Agent submits the token, its CurveZMQ public key (A_{pub}), communication public key ($Comm_{pub}$) and metadata for device registration to server's RESTful API.
7. The Deployment Manager verifies the token and registration document, if valid:
   i. Invalidates the token and registers the device.
   ii. Signs the communication public key ($Comm_{pub}$), which becomes a certificate ($Comm_{cert}$). The algorithms for certificate signing are provided by libsodium[21] based on Ed25519[22].
   iii. Responds with its CurveZMQ public key (M_{pub}) and the communication certificate ($Comm_{cert}$).
8. The Deployment Agent contacts the server over ZeroMQ. They establish a secure channel using CurveZMQ.

Ephemeral key (token)

The registration of a device is made possible by issuing a key on the server and providing it on the device. The design behind the key generation had to address not only the security but also the usability. The user is meant to transfer this key in offline settings and in most cases enter it by hand. Thus, the key had to be short, readable, and writable while remaining secure. Each key can only be used for one registration.

To achieve that, the key is generated as a 12-digit hexadecimal string separated with dashes into three chunks as below:

xxxx-xxxx-xxxx

Some sample generated keys are:

6fa0-3261-ece9
2578-38fe-76a3
b8ac-f877-1eac

The hexadecimal pool and chunking avoid dealing with similar characters (i.e. 0O1Il) and highly improves readability and writability of the keys. This tackles usability concerns, but the security is of paramount importance.

The password entropy is a measurement of unpredictability of a password [25], the formula is:

$$E = \log_2 R^L$$

where E is the key entropy, R is the pool of unique characters, and L is the number of characters in the key. Accordingly, there are $R^L = 16^{12} = 2.8 \times 10^{14}$ possible combinations of the 12-digit hexadecimal key and bits entropy of $E = \log_2 16^{12} = 48$ bits. The entropy bits show that the key is fairly secure. However, with enough time, an attacker would be able to guess a key. To prevent that, the system applies two measures. First, each key is given a 1-week lifetime and will not be valid beyond that. Second, the server is set to throttle key validation requests by enforcing a sequential process. Depending on the sensitivity of the deployments, this can be further throttled down by adding delays or throttling requests based on originating IP addresses. Overall, with this entropy and a 1-week lifetime, an attacker would need to make guesses at least on the scale of 4.6×10^8 per second to succeed, which is not realistic in an online scenario.

4.4 Graphical User Interface

The Deployment Tool proposes a graphical interface developed in three phases. The first phase built the initial

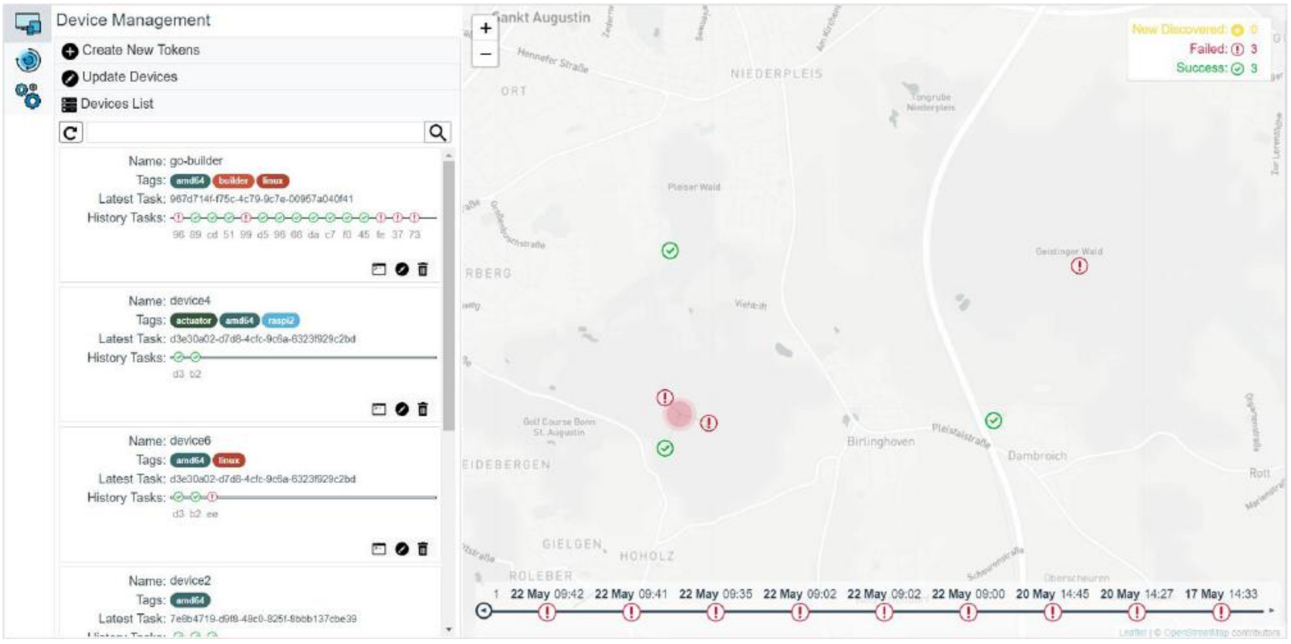
[20] https://libsodium.gitbook.io/doc/public-key_cryptography/authenticated_encryption
[21] https://libsodium.gitbook.io/doc/public-key_cryptography/public-key_signatures
[22] https://ed25519.cr.yp.to/

idea based on user requirements and presented high-level concepts in the form of a low-fidelity prototype. Five domain experts participated in formative evaluations to validate the usefulness of the concepts. The user feedback was applied successively into a medium-fidelity prototype. The same group of users evaluated this prototype and provided feedback to early refinements. The final phase resulted in a high-fidelity prototype presented here. The GUI uses the external APIs of the Deployment Manager.

The GUI is implemented in JavaScript using Vue.js framework[23]. It consists of two main sections for device and deployment management. In both sections, a 2D map of the environment is shown to support different deployment functionalities. The different components of the GUI are presented below.

Figure 12 shows the device management view. On the left, the figure shows device list as expanded. The list of devices is in form of tiles each showing the name, tags, latest task ID, and task history. The list can be filtered by device names or tags (Figure 15). On the bottom of each time, the first button opens a small terminal window for executing commands on devices (Figure 16), the second button is for editing meta data about devices (Figure 17), and the third button is to remove the device from the registry. Apart from the device list, the user can choose to create new tokens (Figure 13) or update devices in bulks (Figure 14).

On the right side of Figure 12, a 2D map shows the devices based on their latitude and longitude. Other forms of localization (e.g. for indoor maps) may be added in future. The map uses green or red icons for each device to show the last deployment status. Devices that are close to each other may be clustered depending on the zoom level. In the figure, the clustered devices are expanded. Below the map, a timeline shows the history of deployments. More information is shown by selecting icons of devices or timeline.



Name: go-builder
Tags: amd64 builder linux
Latest Task: 967d714f-f75c-4c79-9c7e-00957a040f41
History Tasks: 96 89 cd 51 99 d5 96 66 da c7 f0 45 fe 37 73

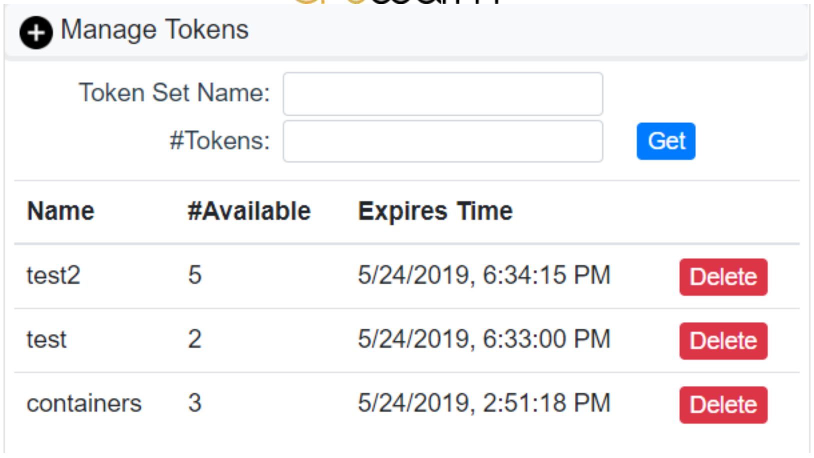
Name: device4
Tags: actuator amd64 raspi2
Latest Task: d3e30a02-d7d8-4cfc-9c6a-6323f929c2bd
History Tasks: d3 b2

Name: device6
Tags: amd64 linux
Latest Task: d3e30a02-d7d8-4cfc-9c6a-6323f929c2bd
History Tasks: d3 b2 ee

Name: device2
Tags: amd64
Latest Task: 7e8b4719-d9f8-49c0-825f-8bbb137cbe39

Figure 12. Device Management view. Device List is expanded, showing the registered devices. The map shows the devices based on their location and their status.

[23] https://vuejs.org/

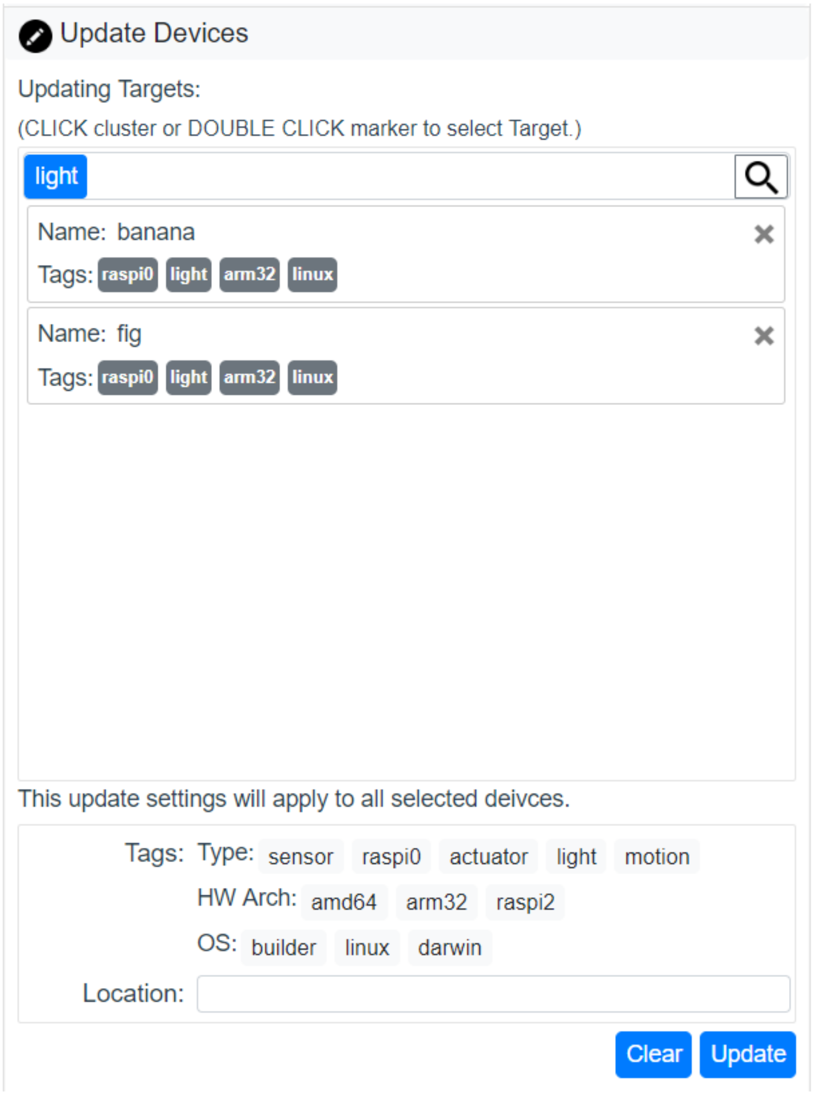

Manage Tokens

Token Set Name:

#Tokens:

Name	#Available	Expires Time	
test2	5	5/24/2019, 6:34:15 PM	Delete
test	2	5/24/2019, 6:33:00 PM	Delete
containers	3	5/24/2019, 2:51:18 PM	Delete

Figure 13. Managing tokens for device registration.


Update Devices

Updating Targets:

(CLICK cluster or DOUBLE CLICK marker to select Target.)

light

Name: banana
Tags: raspi0 light arm32 linux

Name: fig
Tags: raspi0 light arm32 linux

This update settings will apply to all selected deivces.

Tags: Type: sensor raspi0 actuator light motion
HW Arch: amd64 arm32 raspi2
OS: builder linux darwin

Location:

Figure 14. Batch update of meta information about devices.

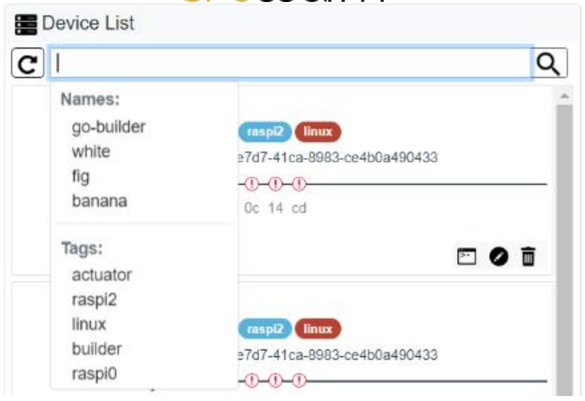
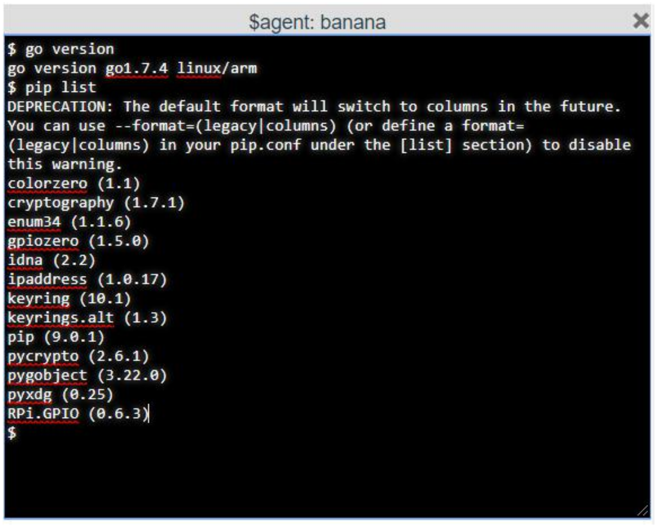
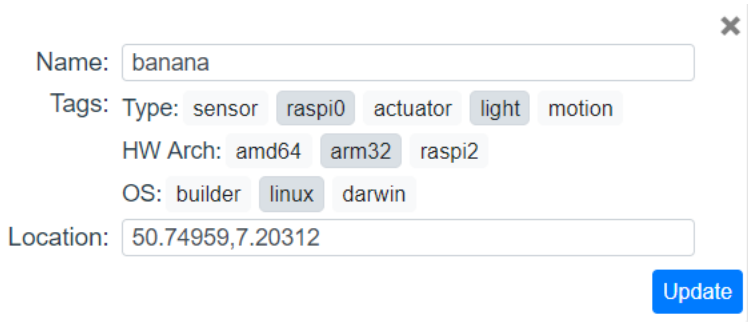


Figure 15. Filtering list of devices by names and tags.



```
$agent: banana
$ go version
go version go1.7.4 linux/arm
$ pip list
DEPRECATION: The default format will switch to columns in the future.
You can use --format=(legacy|columns) (or define a format=
(legacy|columns) in your pip.conf under the [list] section) to disable
this warning.
colorzero (1.1)
cryptography (1.7.1)
enum34 (1.1.6)
gpiozero (1.5.0)
idna (2.2)
ipaddress (1.0.17)
keyring (10.1)
keyrings.alt (1.3)
pip (9.0.1)
pycrypto (2.6.1)
pygobject (3.22.0)
pyxdg (0.25)
RPi.GPIO (0.6.3)
$
```

Figure 16. Terminal window for a device.

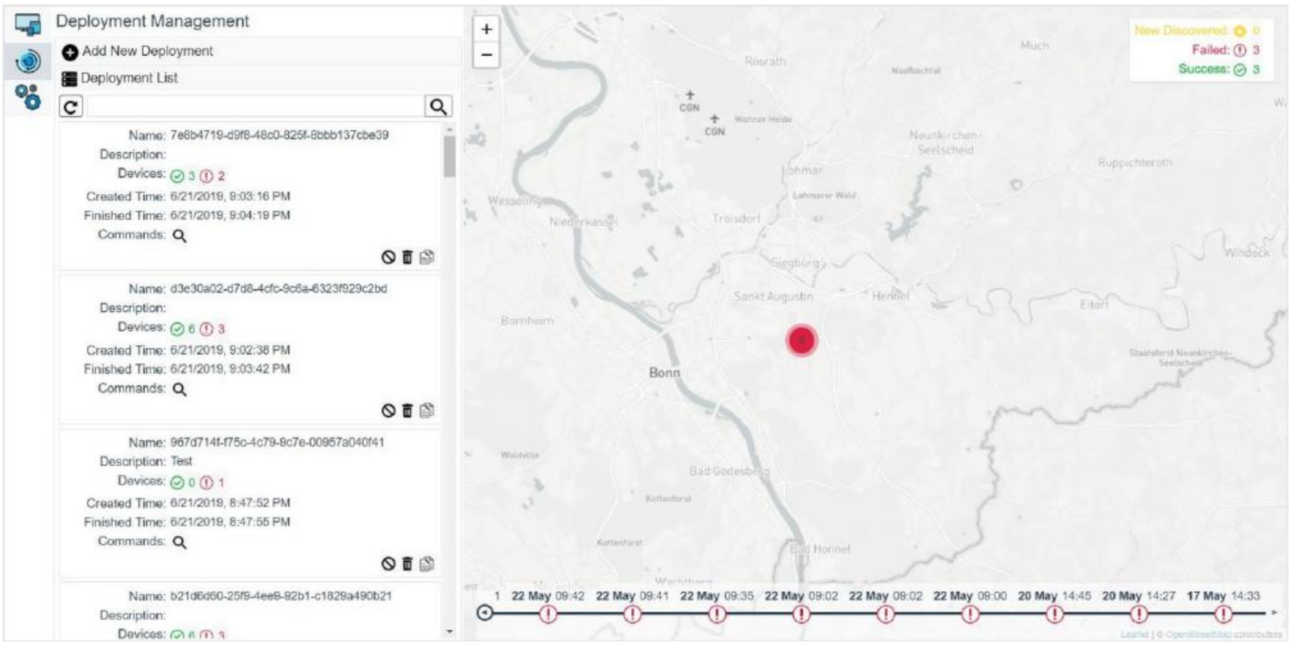


Name:	banana
Tags:	Type: sensor raspi0 actuator light motion
	HW Arch: amd64 arm32 raspi2
	OS: builder linux darwin
Location:	50.74959,7.20312

Figure 17. Updating meta data of devices. This tile opens after selecting the edit button on a device.

Figure 18 shows the deployment view. The left side of the figure shows the deployment pane, with deployment list expanded. This list shows a summary of deployments in form of tiles. Each tile displays name, description, total associated devices and their current status, time of deployment, and few buttons to show the configuration (Figure 19), stop the deployment if it is running, remove the deployment configuration, or duplicate it. Apart from the list, the user can add a new deployment and provide the configuration. Figure 20 shows the expanded view when configurating the deployment. The right side of this figure shows a higher zoom level on the map, with location of the devices in different buildings. In this zoom level, some devices are still grouped because of the proximity. One device is shown individually and has a successful deployment.

After selecting the deploy button in the configuration page, the deployment will start, and the user will be presented with a progress tree and deployment logs; see Figure 21. The progress tree groups device together based on the status. Devices that are failing with the same errors are grouped together so that the user can identify statistics about failures at a glance.



Deployment Management

Add New Deployment

Deployment List

Name: 7e8b4719-d9f8-48c0-825f-8bbb137cbe39
Description:
Devices: ✓ 3 ⓘ 2
Created Time: 6/21/2019, 9:03:16 PM
Finished Time: 6/21/2019, 9:04:19 PM
Commands:

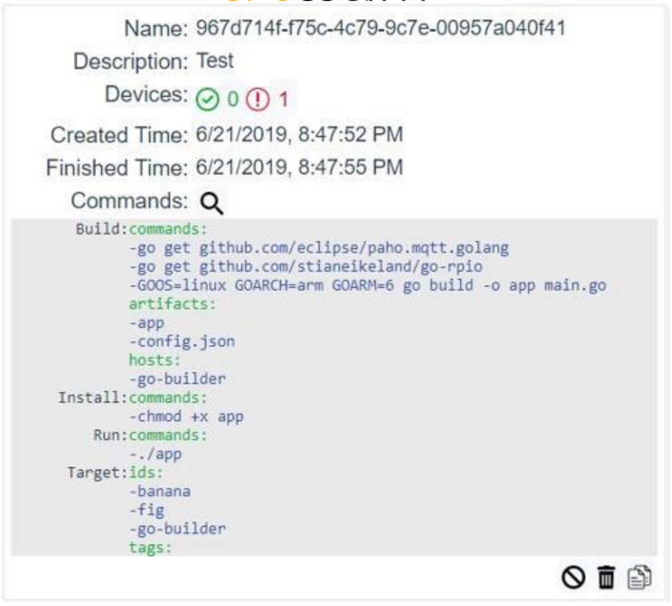
Name: d3e30a02-d7d8-4cfc-9c6a-6323f929c2bd
Description:
Devices: ✓ 6 ⓘ 3
Created Time: 6/21/2019, 9:02:38 PM
Finished Time: 6/21/2019, 9:03:42 PM
Commands:

Name: 967d714f-f75c-4c79-8c7e-00957a040f41
Description: Test
Devices: ✓ 0 ⓘ 1
Created Time: 6/21/2019, 8:47:52 PM
Finished Time: 6/21/2019, 8:47:55 PM
Commands:

Name: b21d6d60-25f9-4ee9-92b1-c1829a490b21
Description:
Devices: ✓ 6 ⓘ 3

New Discovered: 0
Failed: ⓘ 3
Success: ✓ 3

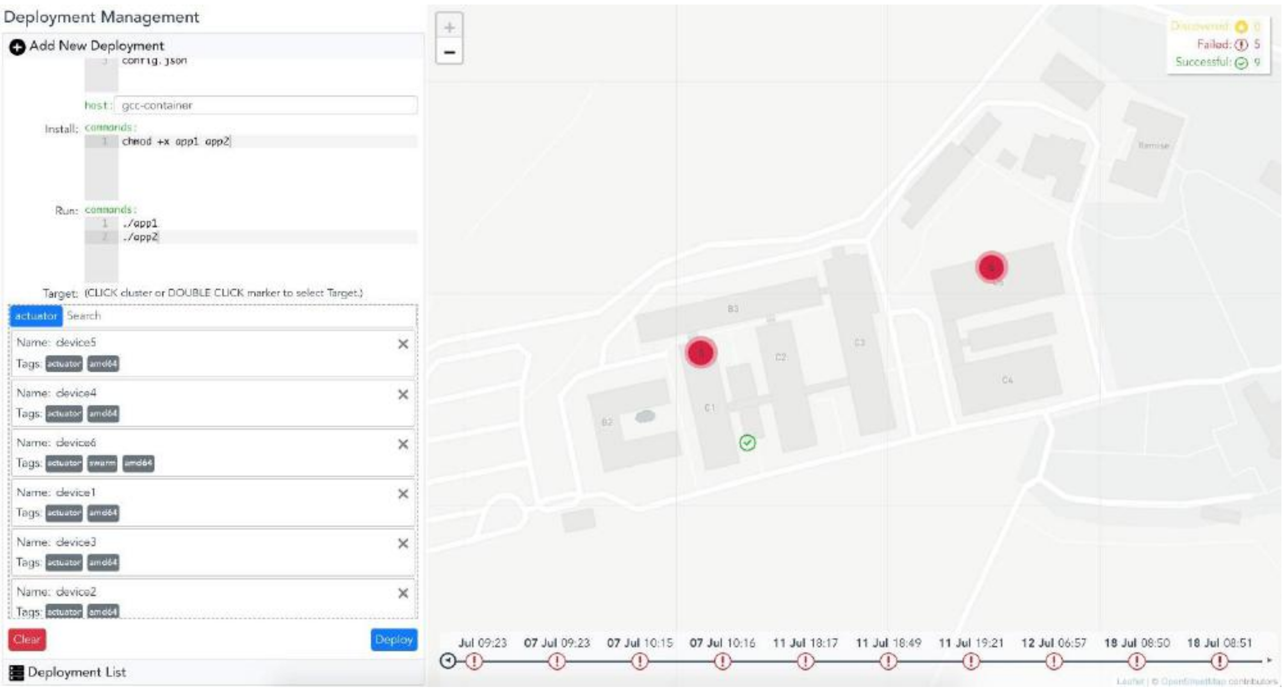
Figure 18. Deployment management view. The deployment list is expanded. On the right, the map shows 6 devices close to each other, some in failure state.



Name: 967d714f-f75c-4c79-9c7e-00957a040f41
Description: Test
Devices: ✓ 0 ⓘ 1
Created Time: 6/21/2019, 8:47:52 PM
Finished Time: 6/21/2019, 8:47:55 PM
Commands:

```
    Build:commands:
          -go get github.com/eclipse/paho.mqtt.golang
          -go get github.com/stianeikeland/go-rpio
          -GOOS=linux GOARCH=arm GOARM=6 go build -o app main.go
          artifacts:
          -app
          -config.json
          hosts:
          -go-builder
  Install:commands:
          -chmod +x app
      Run:commands:
          -./app
   Target:ids:
          -banana
          -fig
          -go-builder
          tags:
```

Figure 19. Expanded deployment configuration.



Deployment Management

Add New Deployment

```
    config.json

    host: gcc-container
Install: commands:
    chmod +x app1 app2

Run: commands:
    ./app1
    ./app2
```

Target: (CLICK cluster or DOUBLE CLICK marker to select Target.)

actuator Search

Name: device5
Tags: actuator amd64

Name: device4
Tags: actuator amd64

Name: device6
Tags: actuator swarm amd64

Name: device1
Tags: actuator amd64

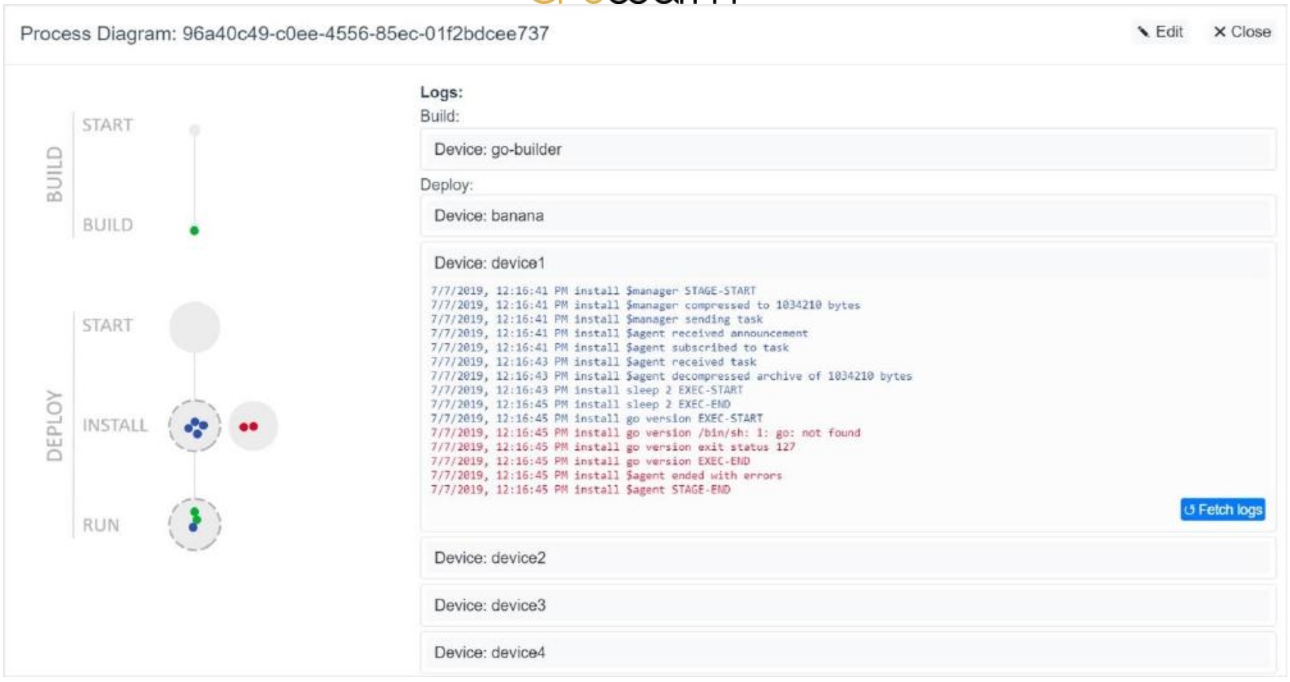
Name: device3
Tags: actuator amd64

Name: device2
Tags: actuator amd64

Clear Deploy

Deployment List

Figure 20. Configuring a deployment for several devices.


Process Diagram: 96a40c49-c0ee-4556-85ec-01f2bdcee737

Logs:
Build:

Device: go-builder

Deploy:

Device: banana

Device: device1

```
7/7/2019, 12:16:41 PM install $manager STAGE-START
7/7/2019, 12:16:41 PM install $manager compressed to 1034210 bytes
7/7/2019, 12:16:41 PM install $manager sending task
7/7/2019, 12:16:41 PM install $agent received announcement
7/7/2019, 12:16:41 PM install $agent subscribed to task
7/7/2019, 12:16:43 PM install $agent received task
7/7/2019, 12:16:43 PM install $agent decompressed archive of 1034210 bytes
7/7/2019, 12:16:43 PM install sleep 2 EXEC-START
7/7/2019, 12:16:45 PM install sleep 2 EXEC-END
7/7/2019, 12:16:45 PM install go version EXEC-START
7/7/2019, 12:16:45 PM install go version /bin/sh: 1: go: not found
7/7/2019, 12:16:45 PM install go version exit status 127
7/7/2019, 12:16:45 PM install go version EXEC-END
7/7/2019, 12:16:45 PM install $agent ended with errors
7/7/2019, 12:16:45 PM install $agent STAGE-END
```

Device: device2

Device: device3

Device: device4

Figure 21. Progress tree and logs for an active deployment.

5 Conclusion

This report elaborated the final design and implementation of CPSwarm Bulk Deployment Tool. It provided an overview of technical requirements gathered throughout the project as well as a brief analysis of most relevant work. It then introduced four design factors for a suitable deployment tool (usability, efficiency, practicality, security) addressing most of the user needs. Based on the given factors, the design was presented considering strengths of existing solutions and most relevant shortcomings. The implementation was described from a high-level point of view, describing different interfaces and important flows. Overall, this document along with the project source code and documentation can be used to learn about the tool, perform bulk deployment in laboratory environments, or to develop it further.

Future work may be dedicated to the following:

- Evaluate the robustness of the system in long term deployments in realistic and large-scale setups.
- Design and evaluate P2P deployment strategies for highly scalable package distribution. Decentralized package distribution may improve large-scale deployments by saving bandwidth and reducing transfer time.
- Validate the effectiveness of proposed graphical layout in large-scale and bulky deployments.

Acronyms

Acronym	Explanation
API	Application Programming Interface
CLI	Command-line Interface
CPS	Cyber-Physical System
DDS	Data Distribution Service
DNS	Domain Name Server
FQDN	Fully Qualified Domain Name
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
NAT	Network Address Translation
NAT	Network Address Translation
OTA	Over-the-Air
PGM	Pragmatic General Multicast
REST	Representational State Transfer
SDK	Software Development Kit
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
XMPP	Extensible Messaging and Presence Protocol

List of figures

References

[1] "Software deployment," [Online]. Available: https://en.wikipedia.org/wiki/Software_deployment. [Accessed 17 08 2018].

[2] L. Columbus, "Roundup Of Internet Of Things Forecasts And Market Estimates, 2016," 27 11 2016. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/#4fb0eefe292d. [Accessed 31 12 2017].

[3] M. Shavit, A. Gryc and R. Miucic, "Firmware Update Over The Air (FOTA) for Automotive Industry," SAE International, 2007.

[4] P. L. Skan, "Method for over-the-air firmware update of NAND flash memory based mobile devices". Patent US7698698 B2, 13 4 2010.

[5] Chef Software, Inc, "Chef - Automate IT Infrastructure," [Online]. Available: https://www.chef.io/chef/. [Accessed 31 12 2017].

[6] Puppet, "Puppet: Deliver better software, faster," [Online]. Available: https://puppet.com/. [Accessed 31 12 2017].

[7] Red Hat, Inc., "Ansible: Automation for everyone," [Online]. Available: https://www.ansible.com/. [Accessed 16 08 2018].

[8] Red Hat, Inc., "Red Hat Ansible Tower," [Online]. Available: https://www.ansible.com/products/tower. [Accessed 16 08 2018].

[9] "SaltStack Documentation," [Online]. Available: https://docs.saltstack.com/. [Accessed 23 08 2018].

[10] SaltStack, "A FRESH LOOK AT SALTSTACK," 06 06 2018. [Online]. Available: https://saltstack.com/a-fresh-look-at-saltstack/. [Accessed 23 08 2018].

[11] Zabbix LLC, "Zabbux," [Online]. Available: https://www.zabbix.com/. [Accessed 31 12 2017].

[12] Nagios Enterprises, LLC, "Nagios," [Online]. Available: https://www.nagios.org/. [Accessed 31 12 2017].

[13] Mender, "Mender: Over-the-air software updates for embedded Linux," [Online]. Available: https://mender.io/. [Accessed 03 08 2018].

[14] "Chef-client memory usage," [Online]. Available: https://discourse.chef.io/t/chef-client-memory-usage/2319. [Accessed 16 8 2018].

[15] L. Hochstein and R. Moser, Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way., O'Reilly Media, Inc., 2017.

[16] R. Ahmed, "What Is Ansible?," 23 07 2018. [Online]. Available: https://www.edureka.co/blog/what-is-ansible/. [Accessed 17 08 2018].

[17] W. Rowe, 01 04 2018. [Online]. Available: https://searchitoperations.techtarget.com/tip/SaltStack-Enterprise-GUI-features-outreach-Salt-Open-territory. [Accessed 23 08 2018].

[18] F. Tavakolizadeh and H. Zhang, "CPSwarm Deployment Tool," 2018. [Online]. Available: https://github.com/cpswarm/deployment-tool. [Accessed 20 9 2019].

[19] "Go (programming language)," [Online]. Available: https://en.wikipedia.org/wiki/Go_(programming_language). [Accessed 02 08 2018].

[20] K. Maeda, "Performance evaluation of object serialization libraries in XML, JSON and binary formats," *Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP),* 16-18 5 2012.

[21] B. Krebs, "Beating JSON performance with Protobuf," 31 1 2017. [Online]. Available: https://auth0.com/blog/beating-json-performance-with-protobuf/. [Accessed 14 6 2019].

[22] D. Bernstein, "CurveCP: Usable security for the Internet.," 2011. [Online]. Available: http://curvecp. org. [Accessed 7 1 2019].

[23] iMatix, "CurveZMQ - Security for ZeroMQ," [Online]. Available: http://curvezmq.org/page:read-the-docs. [Accessed 5 1 2019].

[24] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," *Public Key Cryptography - PKC 2006,* pp. 207-228, 2006.

[25] D. Pleacher, "Calculating Password Entropy," [Online]. Available: https://www.pleacher.com/mp/mlessons/algebra/entropy.html. [Accessed 1 2 2019].

Annexes

Annex A - Security Workshop - March 26-27th 2018, Budapest

The following is a part of the security workshop results in regards to communication protocols. This analysis will be published as part of future deliverables.

The main goal of this workshop was to define message types (including their fields and attributes) and to try and map these to primitives in the libraries proposed (XMPP, MQTT, DDS and ZMQ) with the end goal of selecting a library to build our communications infrastructure upon.
Based on feedback from the partners responsible for the Deployment Tool and the Monitoring and Configuration Tool, we have identified the basic message types required for these tools to work. Since the basic requirements for propagating events and commands were already discussed during the previous workshop, we could assemble a more-or-less complete list of messages required:

	Reliable	Multicast	Confidential	Authenticated	Authorized	Time sensitive
Event *an event has occurred on one of the swarm members that need to be propagated*	Yes	Yes	Yes	Yes	Yes	Yes
Command *the Monitoring and Configuration Tool has raised a remote event on a specific swarm member*	Yes	No	Yes	Yes	Yes	Yes
Artefact *the Deployment Tool has sent a software artefact that needs to be deployed on the swarm member*	Yes	No	Yes	Yes	Yes	No
Status *the swarm member has made progress deploying the software artefact*	Yes	No	Yes	Yes	No	No
Set / Get *the Monitoring and Configuration Tool has sent a request to get or set the value for a global parameter of the*	Yes	No	Yes	Yes	Yes	No

behavior						
Subscribe / Unsubscribe *the Monitoring and Configuration Tool wants to subscribe to or unsubscribe from updates on a property*	Yes	No	Yes	Yes	Yes	No
Telemetry *the swarm member has sent an update for the value of a property to a subscriber*	No	No	Yes	Yes	No	Yes

Please note that response messages which only include a confirmation that the operation has completed successfully are not included, and that the descriptions in italic are only examples for how such a message might be used.
On a lower level, in order to facilitate discovery and to provide a way for swarm members and workbench tools to keep track of the current composition of the swarm, a discovery mechanism is needed. Additional security functionality – like initial authentication and key exchanges – might also happen as part of the discovery process. The following basic message types are required for discovery to work:

- Discover / Present – unauthenticated discovery
- Join / Welcome – authenticated discovery and join request
- Status – periodic update on presence and key parameters

While the exact implementation is left open for later discussion, it is likely that initial requests would be multicast, while responses would then arrive as unicast messages.

These message types can then be mapped to standard primitives found in the communication libraries surveyed so far:

- Publish – subscribe: Subscribe, Unsubscribe, Telemetry
- Request – reply: Get, Set, Command
- Stream: Artefact, Status
- Dish – antenna: Event, Discovery

It is important to note that while these primitives are the best match for each message type, it is feasible to implement them using a different primitive if required. Based on the requirements established so far in terms of primitives and features, the libraries proposed were evaluated and compared:

	Centralized protocols		Decentralized protocols	
	XMPP	MQTT	DDS	ZMQ
Publish – subscribe	Yes	Yes	Yes	Yes
Request – reply	Yes	No	Yes	Yes
Stream	Yes	Workaround	Workaround	Yes
Dish – antenna	Yes	Workaround	Workaround	Yes
Suitability for mesh networking	Low	Moderate	High	High

Resource usage	High	Moderate	Moderate	Low
Ease of use	Easy	Very easy	Hard	Easy

From a security perspective, each of these protocols can be made secure, with varying difficulty and resource use. XMPP and MQTT, being centralized protocols, can use TLS for authentication and confidentiality. DDS based solutions have built-in proprietary solutions which are usually not compatible across DDS implementations. ZMQ has built-in CURVE based security for all primitives except for dish – antenna, where a custom implementation is required.
In the end, the decision was made to use a decentralized solution – which is more suitable for mesh networking and maps better to the concept of swarm behavior. Of the two decentralized solutions, **ZMQ was chosen** based on its gentler learning curve and better support for the required primitives.