D7.3 - INITIAL BULK DEPLOYMENT TOOL

Deliverable ID	**D7.3**
Deliverable Title	**Initial Bulk Deployment Tool**
Work Package	**WP7 – Deployment Toolchain**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**2018-09-23**
Status	**Final**
Lead Editor	**Farshid Tavakolizadeh (FRAUNHOFER)**
Main Contributors	**Farshid Tavakolizadeh (FRAUNHOFER), Bálint Jánvári (SLAB)**

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.

Internal Review History

Review Date	Reviewer	Summary of Comments
2018-09-18	Davide Conzon (ISMB)	Accepted with minor comments
2018-09-18	Artiza Elosegui (TTTech)	Minor corrections

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" from M13 to M21. The consecutive results until M33 will be reported in "D7.4 – Final 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 of the steps planned for the following project months.

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 and resource-friendly manner.

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 has discovered numerous pain points which are faced during daily operations by the CPSwarm application partners. These issues are gathered continuously and formulated into technical issues. This deliverable refers to the requirements reported until M21. Section 1.1 summarizes the technical requirements that are used as the basis for design and development of the CPSwarm Bulk Deployment Tool.

1.1 Gathered Application Requirements

The following tables summarize the result of requirement elicitation for bulk software deployment as part of "D2.3 – Initial Requirements Reports" [RD.2] and "D2.6 - Initial Lessons Learned and Updated Requirements Report" [RD.3].

[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			
Issue Links:	is extended by	CRD-59	The Deployment Agent shall report the...	Quality Check passed
	is extended by	CRD-78	The Deployment Agent shall use the li...	Quality Check passed
	is extended by	CRD-79	The Deployment Agent shall be respons...	Quality Check passed
	is related to	CRD-60	The communication between the Deploym...	Quality Check passed

	is related to CRD-61 The Deployment Manager shall receive ... Quality Check passed
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.
Issue Links:	extends CRD-58 The Deployment Tool shall deploy arte... Quality Check passed is extended by CRD-60 The communication between the Deplo... Quality Check passed is related to CRD-61 The Deployment Manager shall receive ... Quality Check passed
Rationale:	The status of deployment is required in order to monitor and synchronise software updates (automatically or by operators)
Fit Criterion:	The status of deployment is required in order to monitor and synchronise 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 authorised 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.
Issue Links:	extends CRD-59 The Deployment Agent shall report the... Quality Check passed is included by CRD-73 The Deployment Tool shall implement s... Quality Check passed is part of CRD-67 All communications between the swarm ... Quality Check passed is related to CRD-58 The Deployment Tool shall deploy arte... Quality Check passed
Rationale:	Secure deployments are vital for secure operation of swarms.
Fit Criterion:	All security aspects including authentication, authorisation, 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.
Issue Links:	is related to CRD-59 The Deployment Agent shall report the... Quality Check passed is related to CRD-58 The Deployment Tool shall deploy arte... Quality Check passed
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.
Issue Links:	contains CRD-60 The communication between the Deploym... Quality Check passed
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.
Issue Links:	is included by CRD-73 The Deployment Tool shall implement s... Quality Check passed mentions CRD-75 The Deployment Agent shall verify the... Quality Check passed
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.
Issue Links:	is included by CRD-105 The Deployment Tool shall provide the... Quality Check passed is related to CRD-104 The Deployment Tool shall provide the... Quality Check passed
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.
Issue Links:	is included by CRD-105 The Deployment Tool shall provide the... Quality Check passed is related to CRD-103 The Deployment Tool shall provide the... Quality Check passed
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.
Issue Links:	includes CRD-104 The Deployment Tool shall provide the... Quality Check passed includes CRD-103 The Deployment Tool shall provide the... Quality Check passed
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

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 research and industry purposes, the document only analyzes tools that are open source and free of charge.

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.

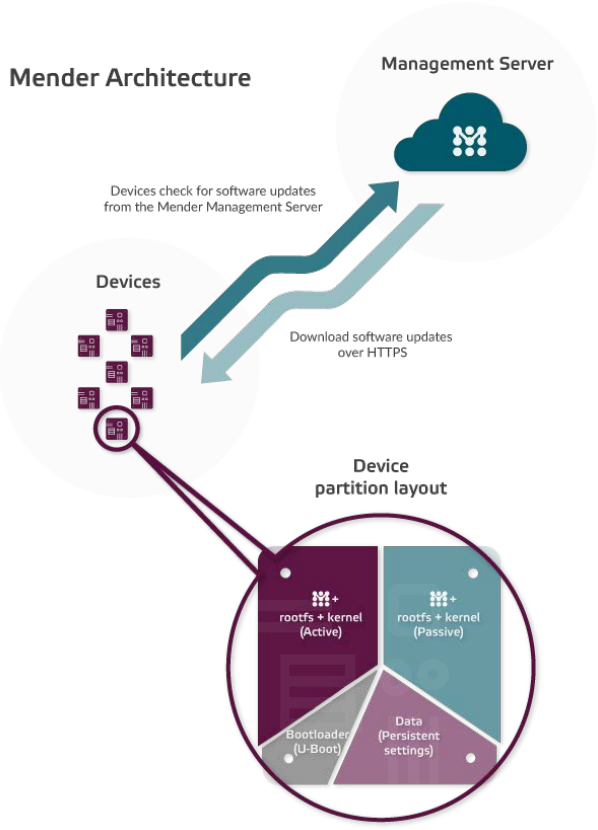

Figure 1. High-level architecture of Mender [13].

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.

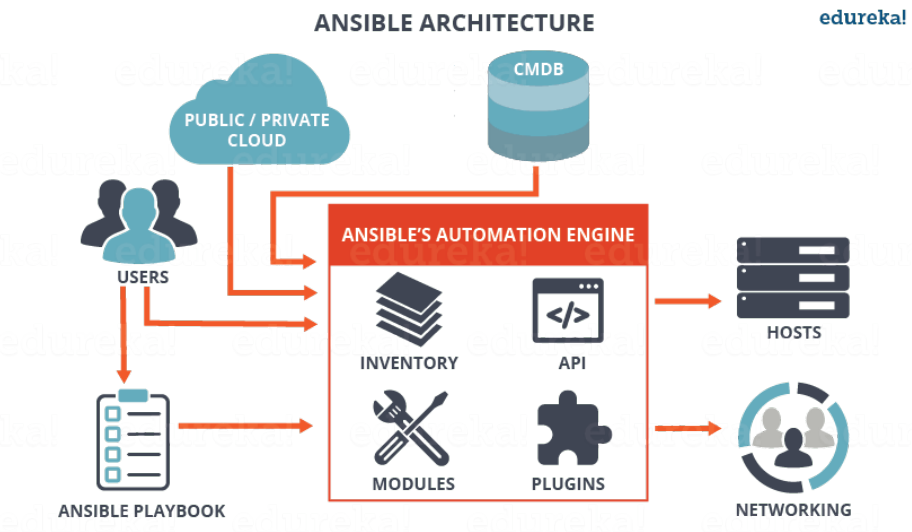

Figure 3. Architecture of Ansible [16].

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, 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

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

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].

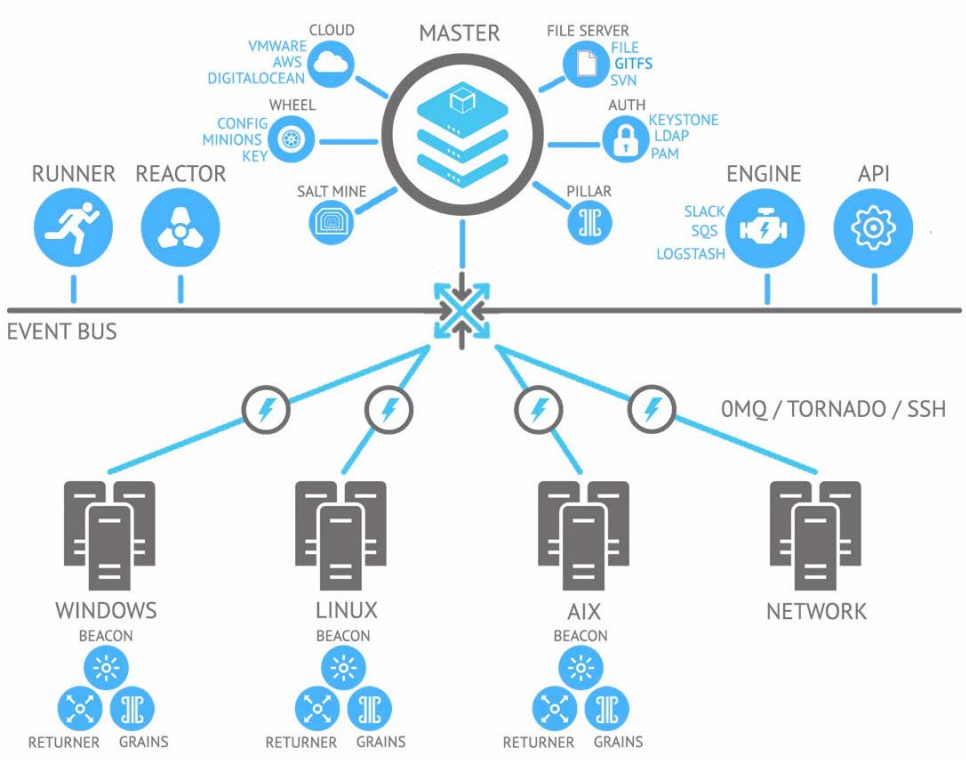

Figure 4. Architecture of Salt [10].

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 reported in [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. (Simplicity)
- 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 take principal role in all iterations of design, implementation, and analysis until the end of the project.

This chapter presents the initial 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 Terminology

Considering simplicity as one of the key factors in the design of the CPSwarm Deployment Tool, the authors emphasize on minimizing the introduction of new terms and instead use familiar terms in software deployment domain. These terms are listed below:

Assembly
Compiling, structuring, signing, and other operations involved during preparation of a package for transfer.

Transfer
Transferring packages to targets. The transfer may involve operations such as compression, encryption, and chunking.

Installation
Placing archive into the right place and preparing it for execution. This may also include installation of dependencies.

Activation
The activity of executing a software after installation. This document uses activation and running interchangeably.

Target
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).

Task Description
A set of instructions and configurations which describe an intended deployment process. This process includes typical deployment steps such as assembly, transfer, installation, testing, and activation. In addition, the Task Description provides information about target devices and logging requirements.

Task

Task or deployment task is an instantiation of a Task Description. The deployment tasks can be categorized into two types depending on their effects on the underlying system: Idempotent tasks are those that make no changes to the host beyond the scope of their task directory. Changes to the system may include installation of dependencies, configuration of daemons, or changes to other files. On the other hand, non-idempotent tasks are those that make permanent changes on the host. Extra care should be given to non-idempotent tasks because they change the state of the system and affect future deployments or other software. Such tasks should ideally include a reversion logic that is triggered when the deployed application turns obsolete.

3.2 Secure Deployment

The CPSwarm Deployment Tool is tasked with ensuring that the software components deployed have authorization to run on the target platform and that the operator is authorized to deploy the software. As a secondary goal, it could also be important in certain use cases to ensure that the proprietary software components deployed remain confidential. To achieve these goals, two basic mechanisms are used at two different points of the develop-and-deploy process, separating the responsibility for issuing authorized software versions and deploying these on live hardware.

After the deployable software package is built, it is signed with the private key of the developer or the organization, protecting the integrity of the package and providing proof that the package can be deployed on the hardware platforms specified within the package. This signature is validated by the Deployment Tool on the device, just before the contents of the package are deployed. In order for a package to pass this validation step, it had to be signed by a trusted developer with a valid signature (with the corresponding public key present in the list of trusted keys), and must have in its metadata explicit permission to be executed on the target hardware platform. Any package that does not meet these requirements will be dropped without any of its components being deployed.

During the deployment process itself, the communication link used to transmit the software package needs to be encrypted and authenticated. For authentication, the operator must present cryptographic proof that it is authorized to perform the operation, which can then be used to negotiate a session-unique encryption key that is used to transfer the package and receive responses generated during deployment.

3.3 Target Selection

The CPSwarm 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*.

3.4 Components

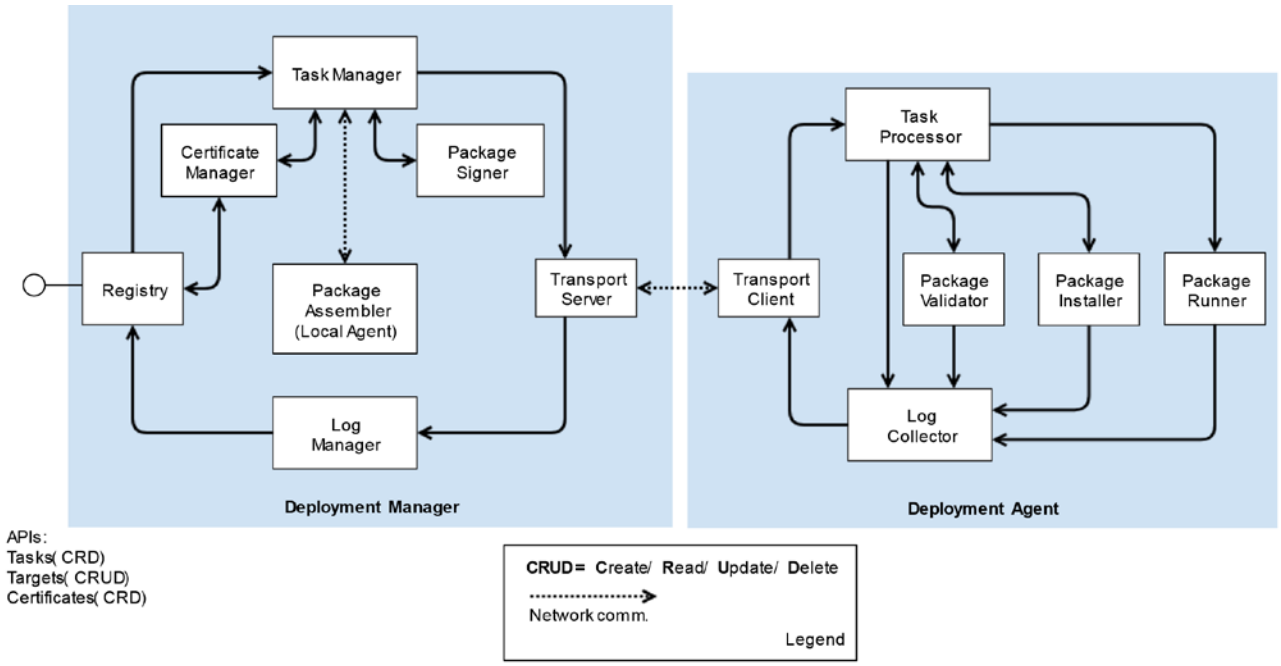

Figure 5. Conceptual diagram of the CPSwarm Deployment Tool

The CPSwarm Deployment Tool follows a client-server architecture with lightweight client component tailored for resource constrained environments and a highly scalable server component. Both 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 over the course of the project and beyond it. 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 CPS with very low footprints. This section briefly introduces these components. The next chapter provides a more technical description of the implemented components.

3.4.1 Deployment Manager

The central component of the Deployment Tool and the main point of interaction for users and command line or graphical interfaces. 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 requests without congestion and overload. This form of vertical scaling enables management 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 deliverable. The Deployment Manager sub-components are described below:

Registry

An information point for keeping the task information and status of every target. The registry exposes APIs which allow management of tasks, targets, and certificates. The registry keeps all tasks in a catalogue along with meta information such as artefact size and matching targets. Targets are stored in another catalogue including meta information about devices, task history, and status of the active task. Detailed logs about individual stages of a deployment can also be queried via the API. Another API provides a way to interact

with Certificate manager to issue, read, and delete certificates for devices. Chapter 4 provides a deeper description of the APIs.

Task Manager
An information processor which receives tasks from Registry and prepares them for transfer to remote devices. The Task Manager maps tasks to targets based on target parameters. It then interacts with Certificate Manager and Package Signer to sign packages. For package assembly, Task Manager submits packages to a local agent which is configured for the intended compilation needs. Final results are sent to the Transport Server.

Certificate Manager
A component that provides utility functions for certificate generation to be used by other components during encryption, authentication, and message signing. The Registry provides an API for users to interact with the Certificate Manager.

Package Assembler
A simplified version of the Deployment Agent, responsible for compilation of packages. Package assembler assists in cross-compilation for other target architectures. It can run on the same hardware, a virtualized, or a remote environment. The compiled packages are subsequently transferred to remote targets. The Package Assembler communicates with Task Manager over network or inter-process communication channels. This enables utilization of remote CPSs or Docker containers as isolated compilation environments.

Package Signer
A component for signing packages that are transferred and validated on target using the Package Validator. Package signing ensures integrity of packages throughout the deployment.

Transport Server
The component enables secure, reliable, and efficient message exchange with the Transport Client 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 based on different protocols, addressing specific use-cases. Every implementation should provide all the required functionalities.

Log Manager
This component processes different kinds of log messages that are sent to the Deployment Manager during target discovery as well as different deployment stages. The component processes and pipes the information to the Registry for storage as well as query responses and status notifications.

3.4.2 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 CPS 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. Logging and security considerations at every step of the deployment assist developers in discovering deployment issues and malicious behaviour. The sub-components of the Deployment Agent are described below:

Transport Client
This component connects to a Transport Server with compatible specifications. 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
A controller in charge of high-level operations and orchestration of other components. Task Processor receives messages from the Transport Client. Signed messages are delegated to Package Validator to ensure their integrity. Furthermore, Task Processor evaluates tasks for compatibility with the hosting target and feasibility of processing and installing them. The result of the evaluation is sent to Log Collector. If a task is accepted, it will be sent to Package Installer and if needed to Package Runner. Task Processor keeps verbose installation and runtime logs in a limited size buffers for debugging and in case requested by the Deployment Manager.

Package Installer
The Package Installer performs all installation steps of a deployment task. This includes writing files to the designated directories, 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 Log Collector. More verbose installation logs are sent to Task Processor 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 are reported to Log Collector but verbose logs are only sent to the Task Processor. Interrupted sub-processes are re-created when the Deployment Agent is restarted. During the restart, Deployment Agent ensures that the package is not tampered with during the interruption.

Log Collector
The counterpart of Log Manager, collecting status and log messages from other components, serializing them and sending the to Transport Client.

Package Validator
All packages are validated before installation and runtime. This component takes a key along with packages or directories as input and verifies content data integrity.

4 Implementation

This chapter present the implementation of CPSwarm Bulk Deployment Tool based on the initial design. The current implementation addresses most of the functional requirements that make the tool usable in an isolated network for in-house deployments. The security components will be implemented in the following months and presented as part of D7.4 – Final Bulk Deployment Tool. Table 1 shows the current implementation status of the components. Regarding the deployment stages, the system is currently stable for transfer, installation, and activation.

Table 1. Implementation status of CPSwarm Deployment Tool as of M21.

Component	Sub-component	Status
Deployment Manager	Registry	Beta
	Task Manager	Stable
	Certificate Manager	-
	Package Assembler	-
	Package Signer	-
	Transport Server	Stable
	Log Manager	Stable
Deployment Agent	Transport Client	Stable
	Task Processor	Stable
	Package Validator	-
	Package Installer	Beta
	Package Runner	Beta
	Log Collector	Stable

The previous chapter explained the four key design factors (simplicity, efficiency, practicality, security) of this deployment tool. These factors also influence the implementation of the system.

The system is developed in Go programming language, an open source compiled language with memory safety, garbage collection, and CPS-style concurrency [18]. The features and the strong built-in libraries of Go make it an ideal language for developing a reliable and efficient program with high parallelization and simple code base.

The following sections describe the implementation starting from external interfaces, diving into the logic of internal components. The overall flow of a single deployment is illustrated in Figure 6 supported here by the description of different APIs.

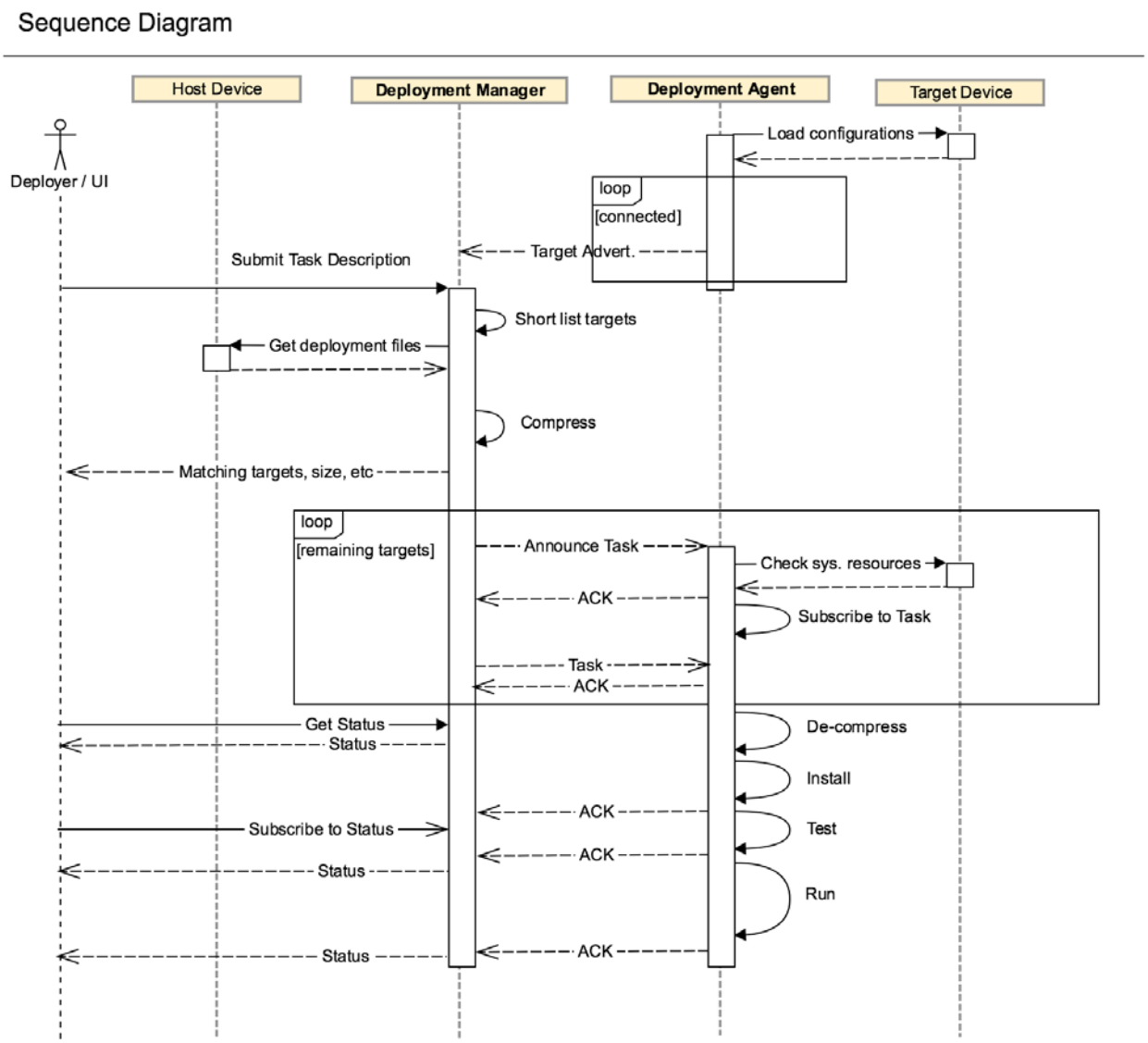

Figure 6. Sequence diagram of deployment on a single target.

4.1 External Interfaces

These interfaces are exposed over the network to be consumed by users and graphical interfaces. As of writing, Targets and Tasks APIs are implemented, leaving Certificates API as future work. Eventually, all APIs would support JSON[7] and YAML[8] as serialization formats. While JSON is highly portable and readable, YAML provides better writability when configuring tasks by hand. The implemented APIs are described as follows.

Targets

A RESTful API offers endpoints to fetch the list of targets, or create, read, update, and delete individual ones. Additionally, an endpoint allows log requests from targets. Log HTTP requests are asynchronous and return immediately. The actual logs arrive at a later time and may be queried using the aforementioned targets endpoints. Table 2 lists the endpoints.

[7] https://en.wikipedia.org/wiki/JSON
[8] https://en.wikipedia.org/wiki/YAML

Table 2. Target API endpoints.

Path	HTTP Method	Request Body	Response Body	Description
/targets	GET	-	List<Target>	Read list of targets
/targets	POST	Target	-	Create a new target
/targets/{id}	GET	-	Target	Read a target
/targets/{id}	PUT	Target	-	Update a target
/targets/{id}	DELETE	-	-	Delete a target
/targets/{id}/logs/{stage}	PUT	-	-	Request logs for a stage

The data model of targets is illustrated in Figure 7. JSON and YAML examples are provided in Appendix A - Sample Task Description.

In addition to the RESTful API, the system exposes a notification channel based on the WebSocket[9] 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. The messages sent on the notification channel follow the Target payload.

Tasks

A RESTful API provides endpoints to list all tasks as well as to create and read them. The API does not offer a way to update or delete 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. The list of endpoints is given in Table 3.

Table 3. Tasks API endpoints.

Path	HTTP Method	Request Body	Response Body	Description
/tasks	GET	-	List<Task>	Read list of tasks
/tasks	POST	Task	-	Create a new task
/tasks/{id}	GET	-	Task	Read a task

The tasks data model is shown in Figure 7. A JSON example is provided in Appendix B - Sample Target List.

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

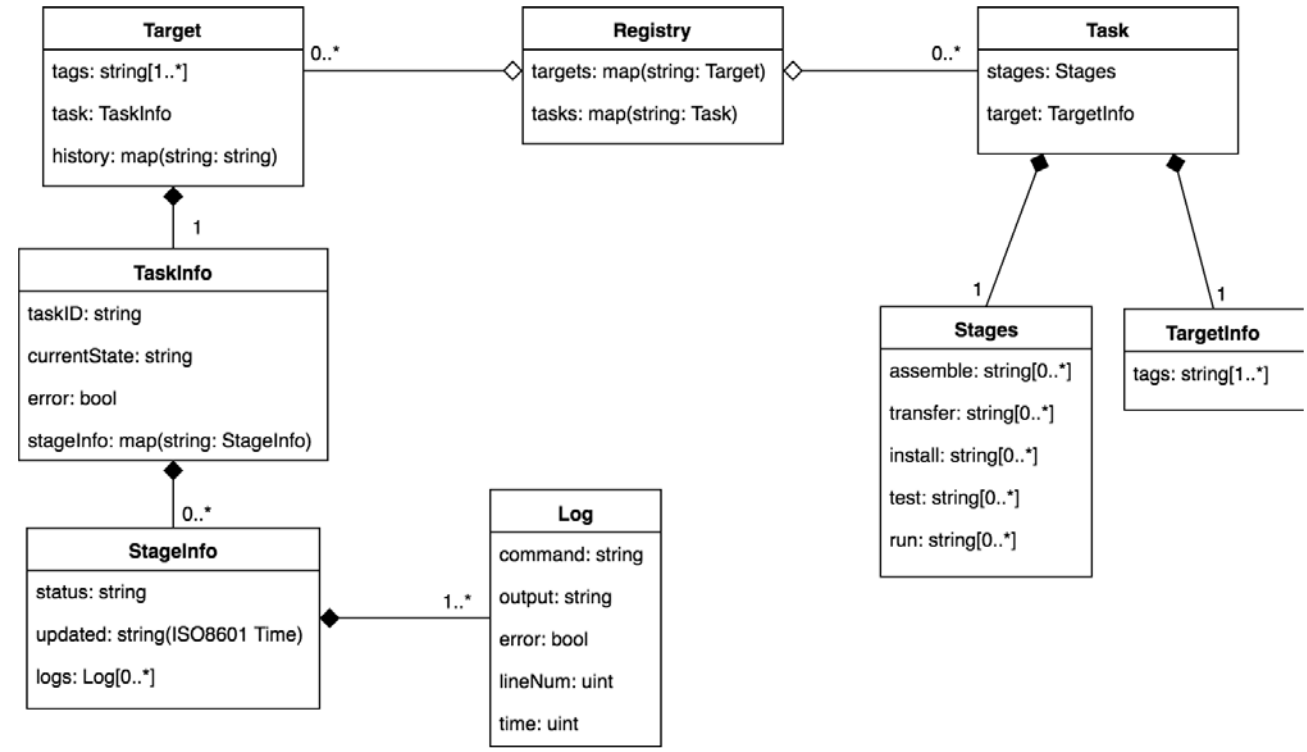

Figure 7. Class diagram of Deployment Manager's data model for the external interface.

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[10], MQTT[11], DDS[12], and ZeroMQ[13] based on a number of factors. This analysis is provided as Annex A - Security Workshop - March 26-27th, 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). 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 [19]. However, the authors consider utilizing the Protobuf[14] protocol to further reduce the serialization bottleneck. This shall reduce message sizes and processing times on manager and agents.

[10] https://en.wikipedia.org/wiki/XMPP
[11] https://en.wikipedia.org/wiki/MQTT
[12] https://en.wikipedia.org/wiki/Data_Distribution_Service
[13] https://en.wikipedia.org/wiki/ZeroMQ
[14] https://en.wikipedia.org/wiki/Protocol_Buffers

Deployment Manager exposes publisher and subscriber interfaces for the following functionalities:

Advertisement and Discovery
The Targets API provides endpoints for creating and removing targets. However, manual management of targets is not feasible in a volatile CPS environment where devices may dynamically join and leave the network. The advertisement and discovery mechanism makes it possible for Deployment Agents to advertise their existence and status to the manager. 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.

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.

5 Conclusion

This report elaborated the initial design and implementation of CPSwarm Bulk Deployment Tool. It provided an overview of technical requirements gathered by M21 as well as a brief analysis of most relevant work. It then introduced four design factors for a suitable deployment tool (simplicity, efficiency, practicality, security) addressing most of the user needs. Based on the given factors, an initial design was presented considering strengths of existing solutions and most relevant shortcomings. The implemented parts of the design were described from a high-level point of view. Overall, this document along with the project source code can be used to learn about the underlying architecture of the tool to perform deployment in isolated environments.

The upcoming project months will be dedicated to the following:

- Detailed design and implementation of features that enable reliable and secure deployments. This will cover different parts of the system ranging from deployment management authorization, to communication, deployment, and execution.
- The ability to assemble packages at the Deployment Manager. This will add cross-compilation and compile-once-for-all capabilities and significantly reduce the installation time.
- 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.
- Gather user requirements for a graphical user interface (GUI). Accordingly, develop the GUI to assist users in deployment configuration and deployment monitoring.

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

List of tables

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

[19] 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.

Appendices

Appendix A - Sample Task Description

YAML

```yaml
stages:
  transfer:
    - package
  install:
    - mv package/* .
    - chmod +x emergencyStop
  run:
    - python -u controller.py
    - ./emergencyStop

target:
  tags:
    - hanger
```

JSON

```json
{
  "stages": {
    "transfer": [
      "package"
    ],
    "install": [
      "mv package/* .",
      "chmod +x emergencyStop"
    ],
    "run": [
      "python -u controller.py",
      "./emergencyStop"
    ]
  },
  "target": {
    "tags": [
      "hanger"
    ]
  }
}
```

Appendix B - Sample Target List

JSON

```json
{
  "drone-1": {
    "Tags": [
      "drone",
      "armhf",
      "hanger"
    ],
    "Task": {
      "ID": "784f439c4034-baff-11e8-b802-3a41140f",
      "CurrentStage": "RUN",
      "Error": false,
      "StageLogs": {
        "Transfer": {
          "Status": "SUCCESS",
          "Updated": "2018-09-14T11:40:35+02:00",
          "Logs": [
            {
              "Output": "received announcement",
```

```
            "Error": false
          },
          {
            "Output": "received task",
            "Error": false
          },
          {
            "Output": "stored artifacts",
            "Error": false
          }
        ]
      },
      "Install": {
        "Status": "SUCCESS",
        "Updated": "2018-09-14T11:40:35+02:00",
        "Logs": [
          {
            "Command": "mv package/* .",
            "Output": "exit status 0",
            "Error": false,
            "LineNum": 1,
            "Time": 1536918035
          },
          {
            "Command": "chmod +x emergencyStop",
            "Output": "exit status 0",
            "Error": false,
            "LineNum": 1,
            "Time": 1536918035
          }
        ]
      },
      "Run": {
        "Status": "SUCCESS",
        "Updated": "2018-09-14T11:47:58+02:00"
      }
    }
  },
  "History": {
    "784f439c4034-baff-11e8-b802-3a41140f": "RUN-SUCCESS"
  }
},
"drone-2": {
  "Tags": [
    "hanger",
    "drone",
    "armhf"
  ],
  "Task": {
    "ID": "784f439c4034-baff-11e8-b802-3a41140f",
    "CurrentStage": "RUN",
    "Error": false,
    "StageLogs": {
      "Transfer": {
        "Status": "SUCCESS",
        "Updated": "2018-09-14T11:40:35+02:00",
        "Logs": [
          {
            "Output": "received announcement",
            "Error": false
          },
          {
            "Output": "received task",
            "Error": false
          },
```

```
            {
              "Output": "stored artifacts",
              "Error": false
            }
          ]
        },
        "Install": {
          "Status": "SUCCESS",
          "Updated": "2018-09-14T11:40:35+02:00",
          "Logs": [
            {
              "Command": "mv package/* .",
              "Output": "exit status 0",
              "Error": false,
              "LineNum": 1,
              "Time": 1536918035
            },
            {
              "Command": "chmod +x emergencyStop",
              "Output": "exit status 0",
              "Error": false,
              "LineNum": 1,
              "Time": 1536918035
            }
          ]
        },
        "Run": {
          "Status": "SUCCESS",
          "Updated": "2018-09-14T11:48:01+02:00"
        }
      }
    },
    "History": {
      "784f439c4034-baff-11e8-b802-3a41140f": "RUN-SUCCESS"
    }
  }
}
```

Annexes

Annex A - Security Workshop - March 26-27th, 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.