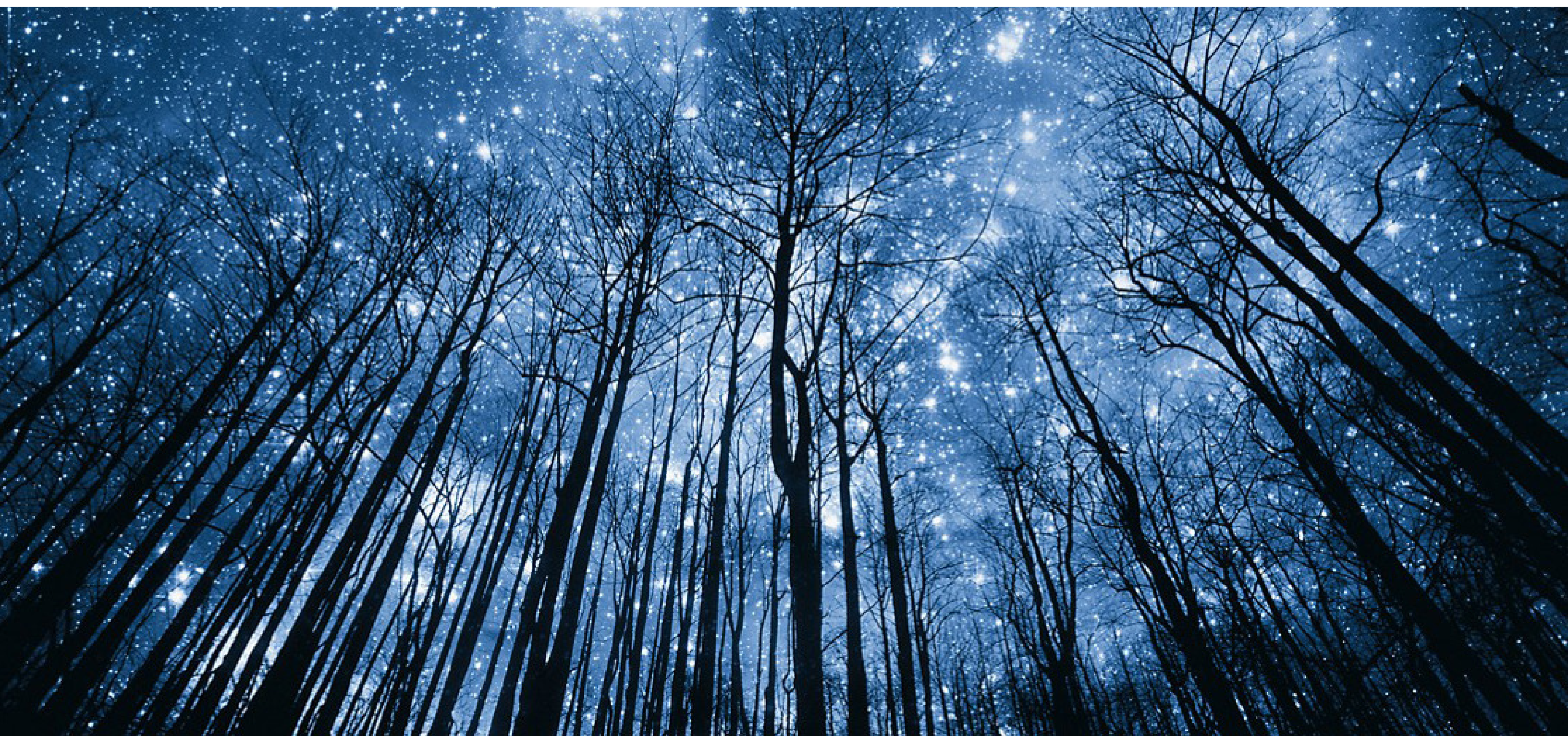
Dell Technologies

A SMART MASSIVE IIOT UPGRADE FRAMEWORK



Mohamed Sohail
Distinguished Member Technical staff
Advisory Consultant, Business Resiliency
Dell Technologies
Mohamed.sohail@dell.com

Mohammad Rafey
Principal Software Engineer, AR Engineering
Dell Technologies
Mohammad_rafey@dell.com

Mohiey Mostafa
Manager, Technical Support
Dell Technologies
Mohiey.mostafa@dell.com

Knowledge Sharing Article
© 2022 Dell Inc. or its subsidiaries.

Dell Technologies Proven Professional

The Dell Technologies Proven Professional Certification program validates a wide range of skills and competencies across multiple technologies and products.

From Associate, entry-level courses to Expert-level, experience-based exams, all professionals in or looking to begin a career in IT benefit from industry-leading training and certification paths from one of the world's most trusted technology partners.

Proven Professional certifications include:

- Cloud
- Converged/Hyperconverged Infrastructure
- Data Protection
- Data Science
- Networking
- Security
- Servers
- Storage
- Enterprise Architect

Courses are offered to meet different learning styles and schedules, including self-paced On Demand, remote-based Virtual Instructor-Led and in-person Classrooms.

Whether you are an experienced IT professional or just getting started, Dell Technologies Proven Professional certifications are designed to clearly signal proficiency to colleagues and employers.

Learn more at www.dell.com/certification

Table of Contents

Introduction 4
Overview 5
Coordination of Massive Devices Upgrade 7
Minimize Offline Requirement 7
Live Migration May Not Work in Some Situations 7
Failure Management 8
Solution Overview 8
Solution Details 9
System Components and Modules 9
Build IoT Device Interaction Pattern based on Weighted Graph 11
Algorithm 20
Conclusion 21
Table of Figures 22
References 22

Disclaimer: The views, processes or methodologies published in this article are those of the authors. They do not necessarily reflect Dell Technologies' views, processes or methodologies.

Introduction

Massive Intelligent Internet of Things (IIoT) devices are being used across a wide spectrum of application domains. Managing that large pool of IIoT devices presents a unique set of challenges in terms of deploying, maintaining, and managing software/firmware upgrades.

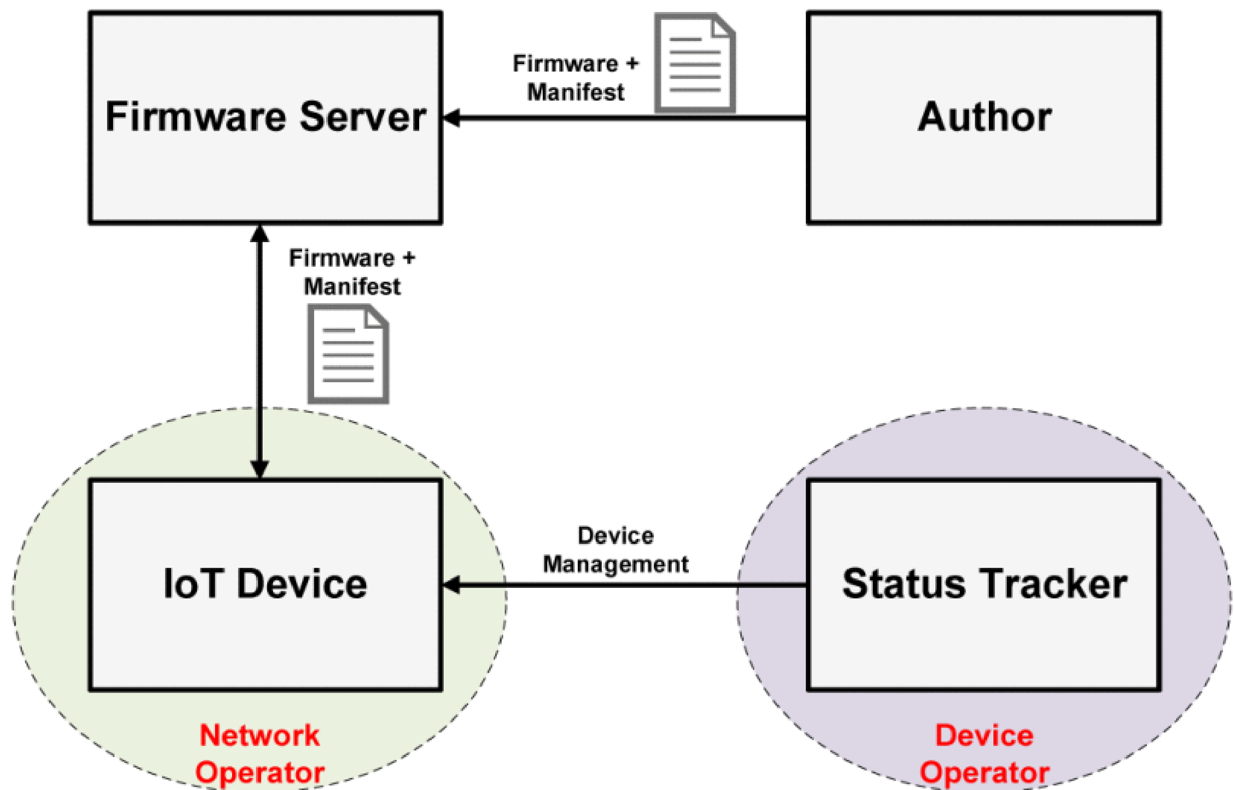


Figure 1:Architecture overview [1]

This Knowledge Sharing article proposes a new and efficient method to overcome many of the issues faced in the current architectural framework.

Some of the benefits are:

- Support software/firmware upgrade granularity as well as leveraging modern container technology.
- Flexible and instant device upgrade or rolling-up upgrade mechanism over a large pool of IIoT devices.
- Smart built-in protection mechanism against upgrade failures and errors.
- Partition devices as domains with dependency management and intelligent coordination.

Overview

As advances continue in Edge and IoT domains, IoT devices are becoming more intelligent and capable in terms of processing power and speed, such as intelligent video cameras, autonomous smart vehicles, medical applications, health welfare, IoT gateway, and many more. A massive number of such IIoT devices have been deployed in smart cities, industrial automation, and instrumentation, which continuously generates and ingests a huge volume of big data for smart decision making via different artificial intelligence (AI) technologies.

IIoT hardware platforms like Raspberry Pi [2] are growing more powerful by the day. Such smart IIoT devices are now getting equipped with modern software stacks like Linux (including Linux microkernel like **ClearLinux**)[3], which is already the prominent operating system deployed on IoT Edge devices and gateways.

A typical large pool of industrial IoT devices could consist of various kinds of devices in the managed domain, having different kinds of hardware built and running various kinds of versions of the operating system, libraries, or applications. These devices may work either independently or interactively such as a few video cameras in the same area from different angles. This entire pool of IoT devices would always require central management, governance, and control of some kind.

Figure 2: Industrial IoT [4]

Managing software upgrades and installation as and when its available is a challenging task over a large pool of IoT devices. Current mechanisms promote/push an upgrade to a specific subset of IoT devices, either in a random fashion or based on their current workload. Often the upgrade happens in some sequential or parallel fashion.

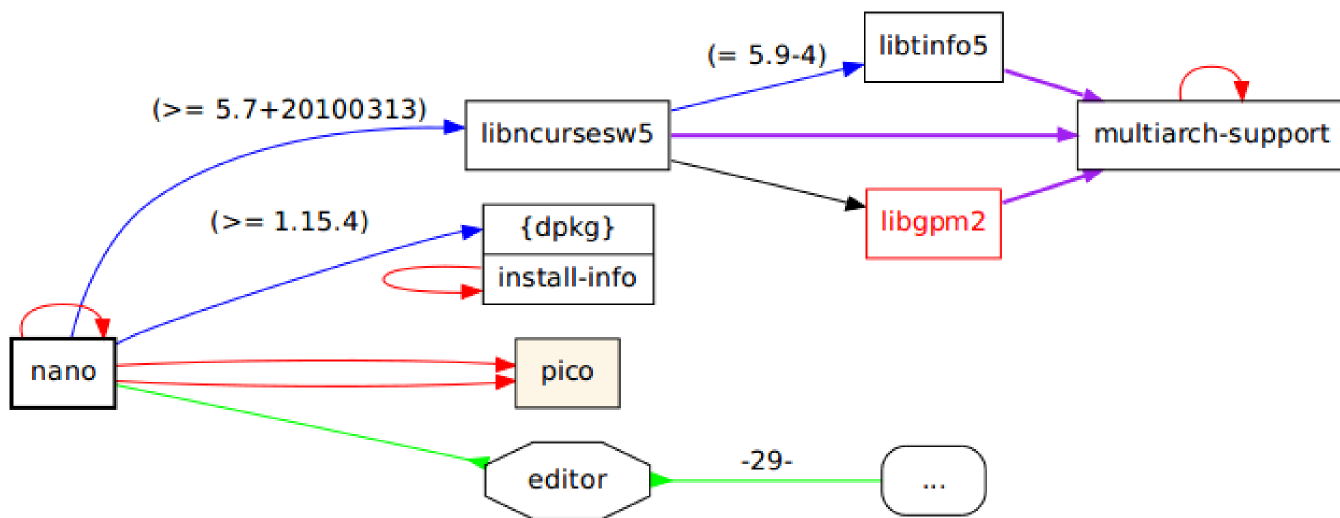


Figure 3: package dependencies in a typical IoT environment

Such mechanisms do not currently have any built-in intelligent features to optimize the deployment schedules systematically. It doesn't take into consideration the current state of each device before pushing an upgrade which is usually resource intensive.

Existing approaches are fundamentally heuristics-based and too costly, risky, and unreliable which results in unexpected outages and negatively affects customer satisfaction. It can adversely impact the performance of such devices and applications it is supporting.

Other detailed problem areas are described below:

Coordination of Massive Devices Upgrade

Devices may have data dependency (such as upstream and downstream) or logical dependency (device must upgrade at first, then can be identified by gateway, etc.), which call for proper dependency management.

Minimize Offline Requirement

A camera or edge may have no standby backup in that area. If it goes offline, data likely cannot be ingested for the monitored area at that moment, thus minimizing offline impact is extremely important.

Live Migration May Not Work in Some Situations

The app usually is tightly bound to the device. Some technology, i.e. App migration from node X to Y with the rolling upgrade, may not work in such circumstances.

Failure Management

In case of an upgrade exception, some intelligent built-in backup mechanism is always needed to roll back.

- Rollbacks must be integrated into the testbed of the firmware development process.
- Basic IoT properties like hostname, networking configuration, and basic features configuration should be stored in a specific file that can be read by any version of the firmware. This practice ensures that despite any rollbacks, the IoT device remains connected and running. An alternative approach is to erase setup and configuration files of the current firmware version and run a new setup process for the IoT device with the old firmware loaded while accessing the previously stored configuration files.

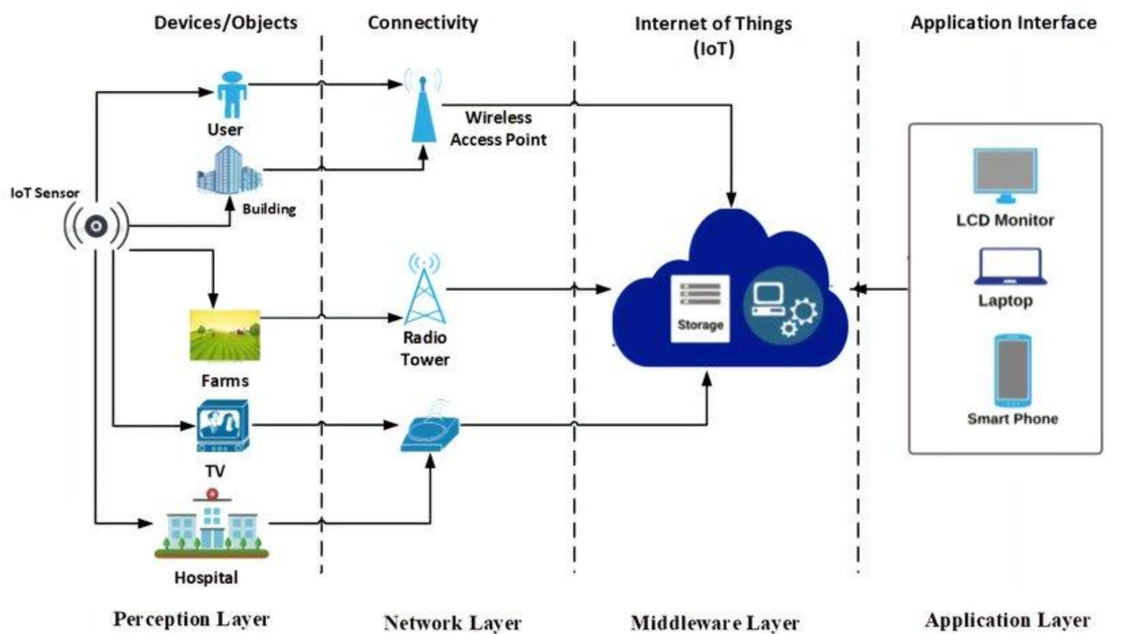


Figure 3: Generic IIoT device layers

Solution Overview

IIoT deployments are supporting the realization of our new digital society. As we now know, any connected device is a favorite target of attackers. Indeed, each device will have to deal with new vulnerabilities and attacks during its lifecycle. Therefore, defining a secure mechanism for software/firmware updates is essential to guarantee security of IoT devices.

This Knowledge Sharing article examines a novel solution that will achieve resiliency solutions for software/firmware updates in IoT devices.

We target a **central managed IIoT environment** where massive IIoT devices are managed, operated, and controlled by a specific single owner identify such as a company, government, etc. rather than many individuals. Typical examples include intelligent video cameras in smart cities, healthcare devices, sensors and edge devices in industrial automation, etc. Mobile phones, home automation, personal automotive cares are typically not our interested targets.

We focus on solving the issue of dependency management when it comes to upgrade and management of IoT devices, especially in massive IIoT, such as:

- App logic upgrade: such as a camera to upgrade the capturing logic or even deep learning model; gateway device to upgrade the management agent.
- Upgrade for performance improvement.
- Critical bug fixing: fix some known critical bugs.
- System-level upgrade (driver, library, even OS): such as for security patch, device interoperation, or system-level performance, etc.

Such IoT devices in the pool can be dependent or independent of each other in regard to their serviceability and current installed software stacks on each device in terms of pre-reg, co-req, and other parameters that are important to build these paths. The proposed method also defines the sequence in which the upgrade should happen.

The system outputs a recommendation relationship sequence for a massive pool of IoT devices, which aims to help the Device Management Tool follow the sequence that would be a hybrid of parallel and sequential pushing of installations or upgrades.

Solution Details

The solution's details that will make it an easy and robust way to handle such a massive amount of IoT devices include:

System Components and Modules

Control Manager – Central Control Manager could be a cluster that monitors and controls all the devices and their software, with a few central metadata stores.

Config Datastore – Keep devices' basic info, i.e. ID, hardware spec, access info, domain info, etc.

Device State Datastore – Device running status details, i.e. health, etc.

Package Store – Well-tested OS images, packages, App container images.

Dependency Graph – Devices' upgrade dependency details.

Gateway – Existing gateway devices that may aggregate data and cache packages.

Edge Devices – Existing devices that run specific tasks or applications on lightweight container technology such as Docker, etc. on modern Linux OS or microkernel Linux.[5] On each such device, an agent is running that collects the device's running status, its software configuration, and versions information.

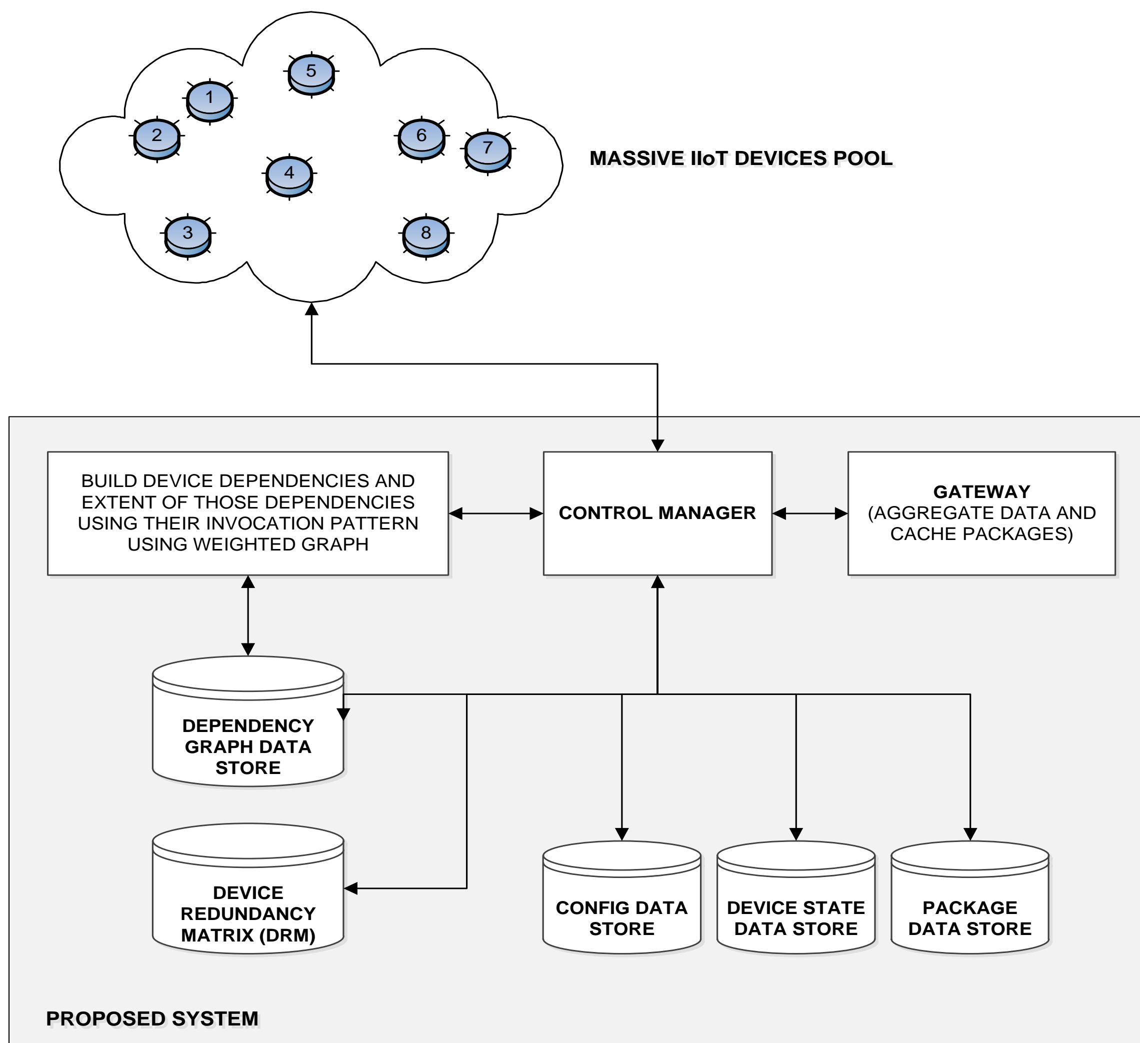


Figure 4: Sample IoT Devices Dependency Weighted Graph

Build IoT Device Interaction Pattern based on Weighted Graph

System models the various devices, and their invocation-dependency patterns as a Directed Weighted Graph with servers/devices as Nodes and their call relationships as Edges.

Edge weights are calculated based on objective function with multiple determining factors like n1 (number of requests), n2 (data in data out), and nm metrics.

It will further be normalized between 0.0 - 1.0 with 0.0 being least active and 1.0 being most active.

Unidirectional Edge weights denote the relative extent of the server's activity, dependency, and relationship among each other.

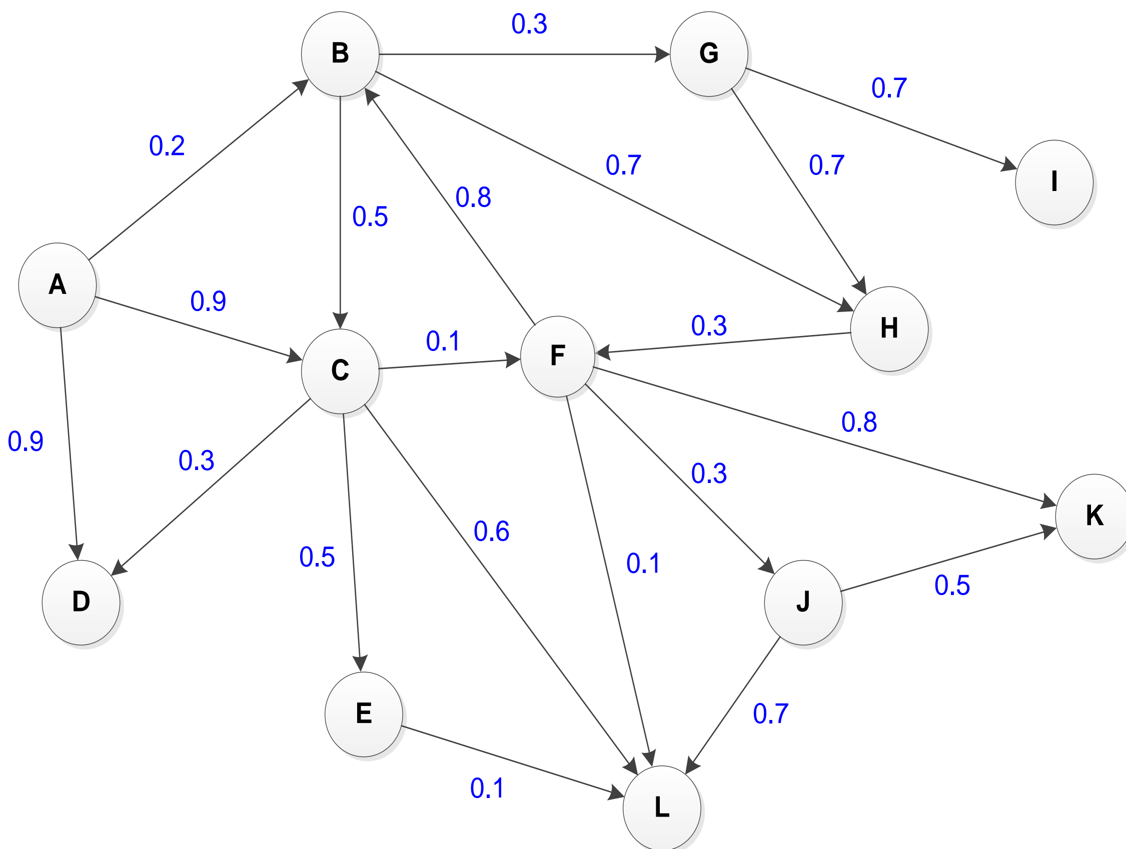


Figure 5: Sample IoT Devices Dependency Weighted Graph.

Build Device Dependency Chain (DDC)

Based on the weighted graph from the previous step, the system identifies a "Server Dependency Chain" by analyzing and computing server similarity scores based on their activity patterns using SimRank.

The server with the least score in the chain is having least dependencies in terms of overall chain activities. The server with the maximum score is having the most dependencies relative to the chain activities.

Formally, it can be expressed as:

$$S(A,B) = \frac{C}{(|I(A)|*|I(B)|)} \sum_{i=1}^{|I(A)|} \sum_{j=1}^{|I(B)|} S(Ii(A), Ij(B)) \quad ..(1)$$

$$S(A,B) = \frac{C}{(|O(A)|*|O(B)|)} \sum_{i=1}^{|O(A)|} \sum_{j=1}^{|O(B)|} S(Oi(A), Oj(B)) \quad ..(2)$$

Where:
C is a constant between 0 and 1.
If there are no IN-neighbors in either A or B i.e. I(A) = Ø or I(B) = Ø, then S(A, B) = 0.

Figure 6:Scoring Equation

Equation 1 computes the IN score and equation 2 computes the OUT score between servers A & B. IN + OUT score combined denote the extent of server activity between A & B.

The server which finally gets the maximum score is placed on top of the "Server Dependency Chain" and then the next scoring server comes and likewise the chain is generated.

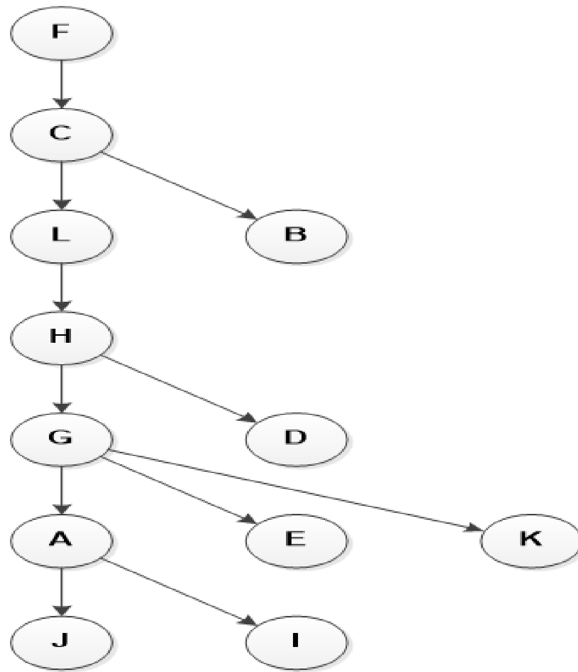


Figure 7: Sample IoT Device Dependency Chain built based on Sample Data

Build Device Redundancy Matrix

A unique configurable Device Redundancy Matrix (DRM) is introduced to the new architecture as part of the fault-tolerant enablement. DRM consists of details of a set of IoT devices and each of the other devices which could act as redundancy or standby for other specified devices.

When one device is being upgraded, another nearby device can be configured to take over the traffic of the upgrading device; hence, device rolling upgrade is supported.

For example, 3 video cameras are set up in a highly secure area. Video 1 is in upgrade mode, video 2 or 3 can be configured to scan a larger area that covers video 1 scope also. If the device has no standby or is not part of DRM, the rolling upgrade will not be supported and an in-place upgrade is a choice.

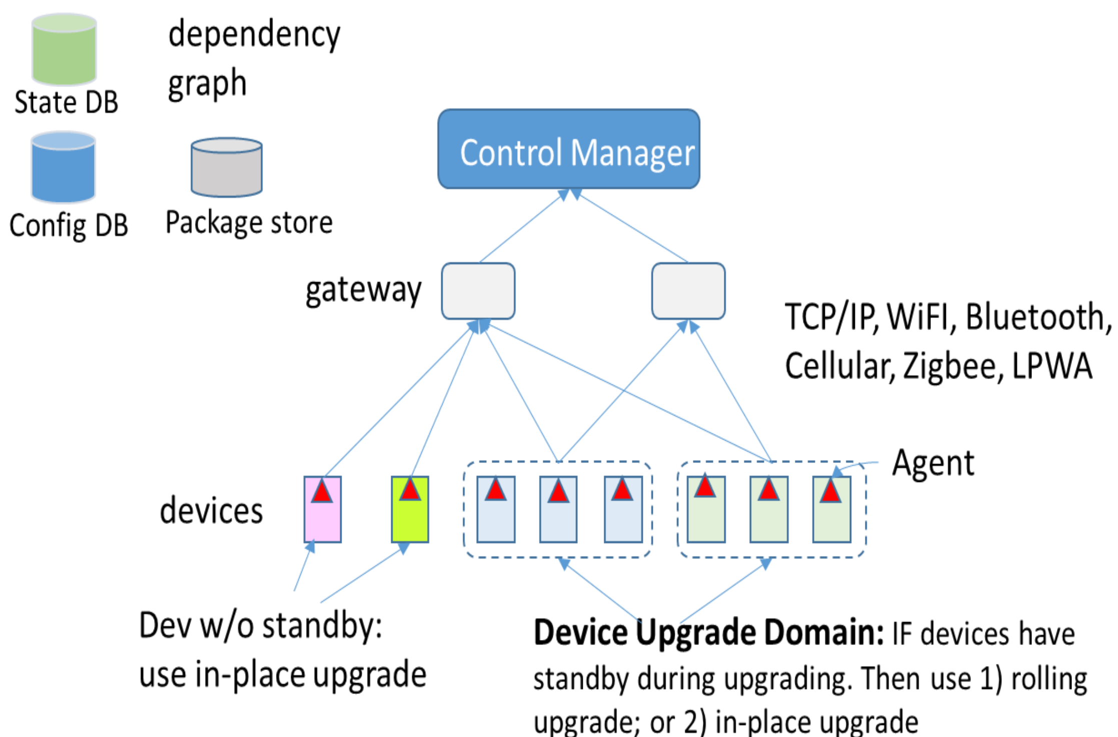


Figure 8: Massive IIoT Software Management Architecture.

Flexible update granularity could be supported, including File: the smallest granularity such as for some configured files.

- Package: package of numbers of App files, lib, etc.

- Container: in cases where App logic could be packed, the device can run Linux and container.
- OS image: the bottom line, everything is refreshed, large data to transfer and potential risk to boot-up failure.

Filter candidate devices and prepare image: query device DB to filter out the candidate device list, check image store to get version prerequisites.

Dependency Coordinator form the dependency graph if there is dependency. Push the image to gateway then to devices, perform an in-place upgrade (if no standby) or rolling upgrade (has standby within upgrade domain). The gateway could act as a temp image caching device to reduce bandwidth consumption over remote access.

The agent keeps working and monitors App healthy info. If an exception occurs, trigger exception handling, such as rollback the upgrade.

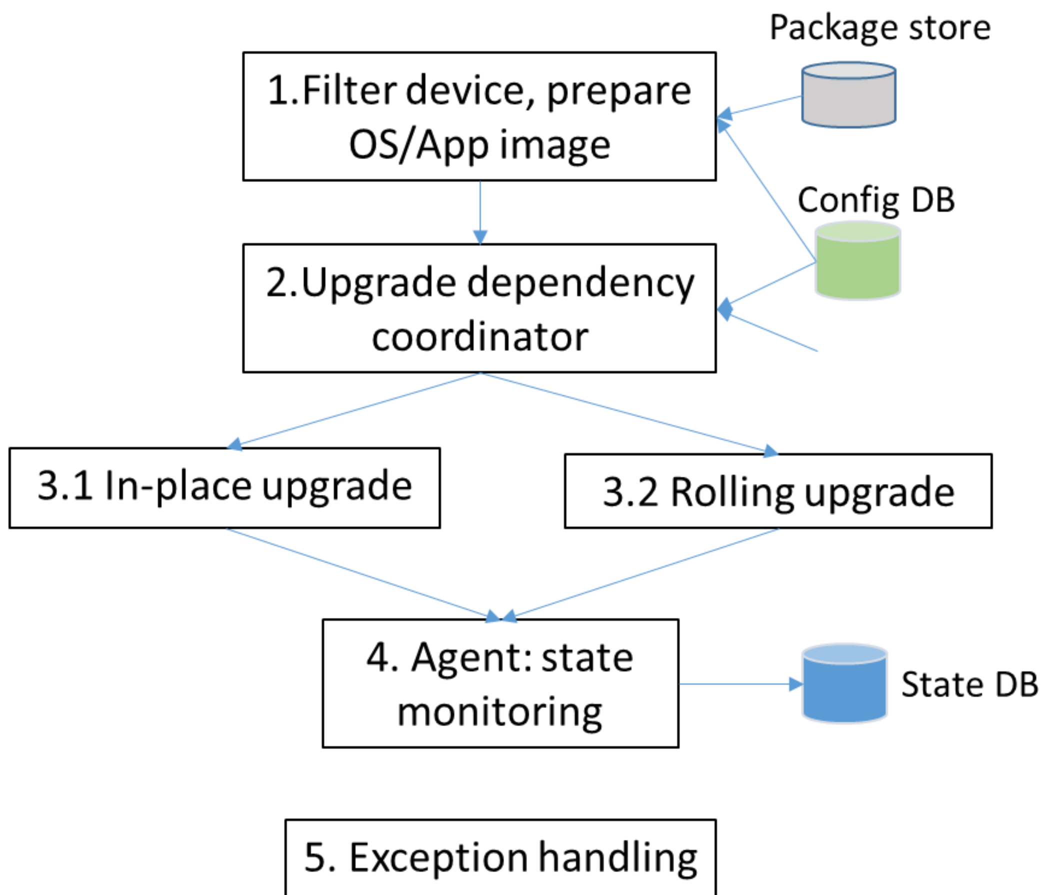


Figure 9: Software Installation/Upgrade Decision Flow

In-place device upgrade steps are shown in Figure 9, where we may leverage modern Linux patch capabilities and container fast launch and container level checkpoint. Modern in-place kernel or container upgrade could be done in dozens of seconds level (see ATC paper in References section).

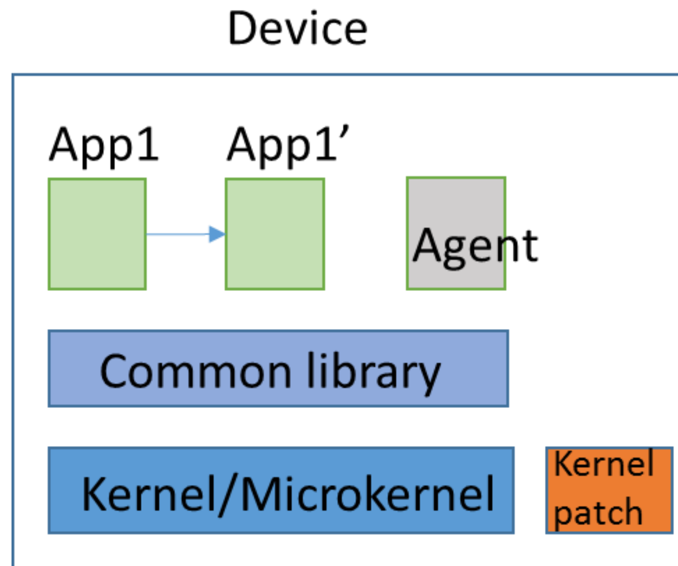


Figure 10: In-Place Upgrade Mechanism.

1. Kernel in-place live upgrade, e.g., kpatch
2. Kernel switch once the live upgrade is done
3. App upgrade: New App container instance (App1)
4. Old App1 in-memory state checkpoint, e.g. CRIU

Rolling upgrade devices in **DRM** domain, assuming two devices in the domain:

1. To upgrade device1, notify device2 and configure App2 to take over (or launch new App1’in device2 depending on the concrete task).
2. Upgrade device1: OS first if necessary then App. May reboot device1 if required.
3. Device 1 is upgraded.
4. Upgrade device2: repeat the above steps.

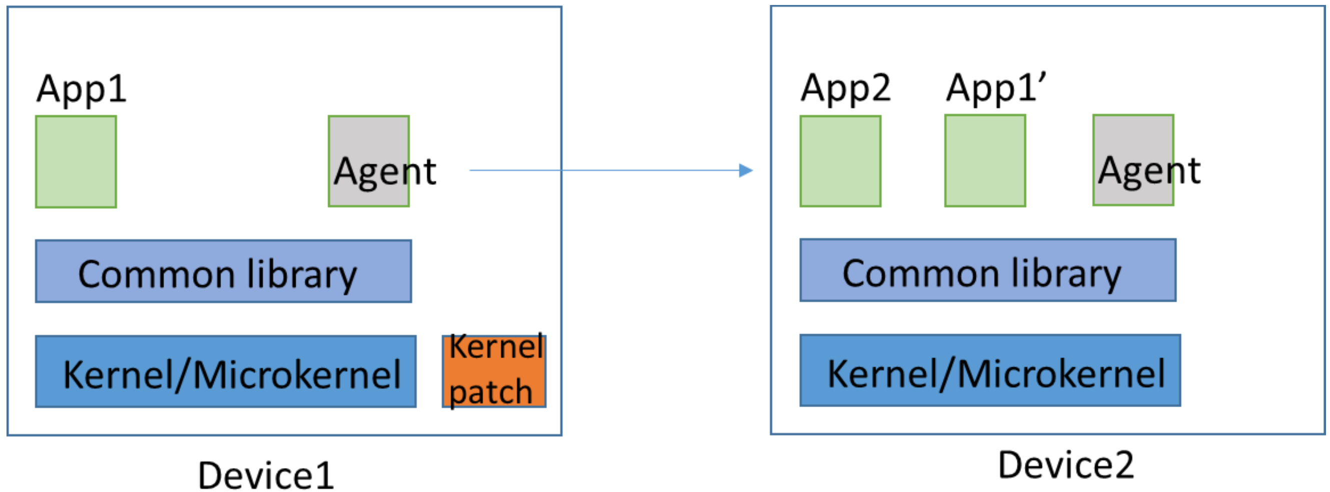


Figure 11: Rolling Upgrade Devices in Upgrade Domain.

Failure Management – One or a few previous package/container/OS images are kept in case of any failure, such as App container launch failure. The agent can monitor the running status, log the event, and launch the old container if configurable; If OS image bootup failure, old images can be kept as standby (Figure 11) or a rescue image for urgent recovery.

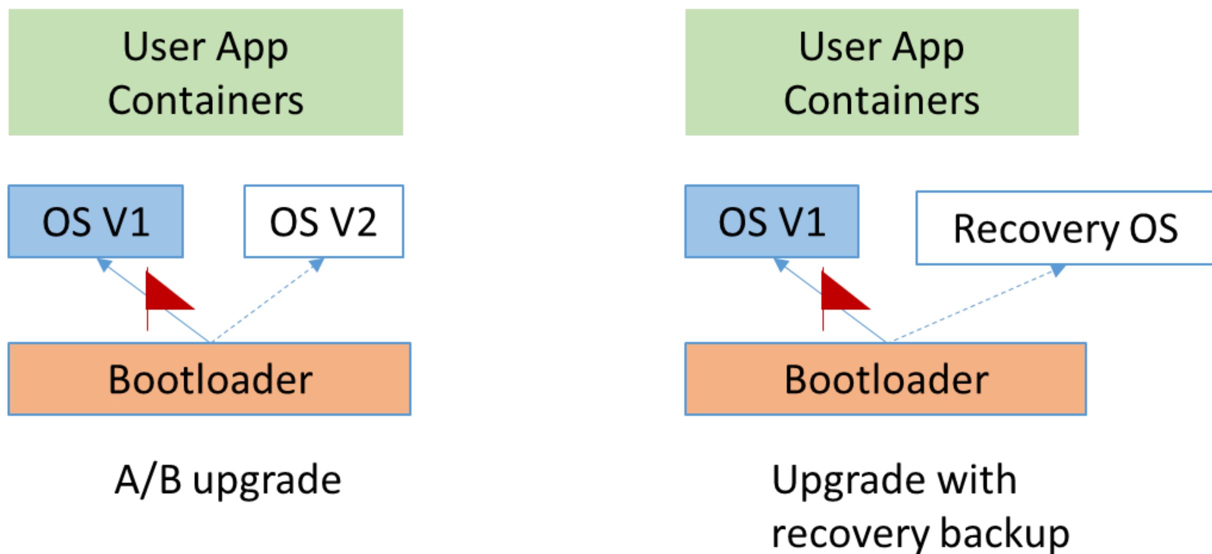


Figure 12 Backup Image During Upgrade.

Workload Prediction of IoT Devices for Deployment Slot Allocation

The system collects workload data metrics on an hourly (or configurable) basis for each device. It then does a Multi-Variate Regression Analysis using Machine Learning techniques, to predict the workload for each slot of the day for each device. Establish an

optimum deployment slot for each device belonging to a chain based on the least workload slot predicted for each device in the entire day.

IOT Device ID	Date Time Slot	Compute Utilization (%)	Memory Utilization (%)	Availability (%)	Response Time (ms)	Storage Utilization (%)
SRD-1	11/7/2020 1:00-2:00	67	75	78	19	65
SRD-1	11/7/2020 2:00-3:00	88	70	88	44	42
SRD-1	11/7/2020 3:00-4:00	50	55	94	10	83
SRD-1	11/7/2020 4:00-5:00	77	44	91	7	54
SRD-1	11/7/2020 5:00-6:00	63	48	65	21	22
SRD-1	11/7/2020 6:00-7:00	55	65	83	75	83
SRD-1	11/7/2020 7:00-8:00	44	42	54	70	54
SRD-1	11/7/2020 8:00-9:00	48	83	65	55	55
SRD-1	11/7/2020 9:00-10:00	35	54	42	83	44
SRD-1	11/7/2020 10:00-11:00	83	65	83	54	48
SRD-1	11/7/2020 11:00-12:00	54	42	42	65	83
SRD-1	11/7/2020 12:00-13:00	43	83	83	42	54
SRD-1	11/7/2020 13:00-14:00	35	54	83	83	65
SRD-1	11/7/2020 14:00-15:00	87	44	54	55	42
SRD-1	11/7/2020 15:00-16:00	83	48	65	44	83
SRD-1	11/7/2020 16:00-17:00	54	75	42	48	70
SRD-1	11/7/2020 17:00-18:00	34	70	83	83	55
SRD-1	11/7/2020 18:00-19:00	65	55	23	54	42
SRD-1	11/7/2020 19:00-20:00	42	83	65	87	83
SRD-1	11/7/2020 20:00-21:00	83	54	65	75	54
SRD-1	11/7/2020 21:00-22:00	54	65	42	70	83
SRD-1	11/7/2020 22:00-23:00	42	42	83	83	54
SRD-1	11/7/2020 23:00-24:00	83	83	54	54	54

Figure 13: Sample Workload Data Details

```
1   import pandas as pd
2   import numpy as np
3   from sklearn.linear_model import LinearRegression
4   from sklearn.metrics import mean_squared_error
5
6   # Read workload captured data.
7   # Please refer the "Workload.xls" to see some sample data.
8   df = pd.read_csv('iot-device-workload-data.csv')
9   df.head()
10
11  # Input data provides date time slot wise workload data on an hourly basis.
12  # System takes into account this captured data and does a multi variate linear regression analysis
13  # to predict the workload for each time slit of the day.
14  y = df['workload-data-server']
15  x = df[['cpu-util','memory-util','availability', 'response-time', 'disc-util']]
16
17  # Define Multi Variate Linear Regression model.
18  linear_regress = LinearRegression()
19
20  # Train model.
21  linear_regress.fit(x,y)
22
23  # Workload pediction.
24  y_pred = linear_regress.predict(x)
```

Figure 14: POC Source Code for Multi-Variate Linear Regression Workload Prediction

Synchronize and Develop Deployment Schedule

In this step, the system takes into consideration the position of each device in the Device Dependency Chain and its workload prediction for the day to determine an optimum deployment time slot and execution sequence schedule.

If a dependency exists, installation or upgrade must run at the given order like a rolling upgrade in proper defined order as per the dependency graph.

Algorithm:

1. For Each IoT Device in the Pool:
   - Select Device whose least workload prediction slot is the earliest in 24 hours.
   - Select the Device that had the least score in the Device Dependency Chain.
   - Combine both the factors and that Device is put on top of the Deployment Schedule.

2. For Each Deployment Schedule in the Pool:
   - Select the first Device from the schedule.
   - Check if the Device has a Redundant Backup in the Device Redundancy Matrix (DRM).
   - If Yes, execute roll upgrade on the currently selected device.
   - If No, execute an in-place upgrade on the currently selected device.

Conclusion

In this article we presented an intelligent, systematic, and programmatic methodology supported by proposed stable, scale-able, and fault-tolerant architecture to manage upgrades across thousands of devices by leveraging SDCUDWG and various other upgrade methods as referenced in figure 6 and 7. To the best of our knowledge, there is no existing framework that offers all the above functionalities in its entirety. The innovative functionalities ensure customer requests can be effectively handled by saving both time and resources.

The updating process for the IoT devices' firmware is a crucial process to guarantee an efficient way for a compliant secure operation of IoT devices throughout their lifecycle. In this Knowledge Sharing article, we illustrated an efficient and tested methodology of some of the main challenges, and how we can smartly address them to realize a comprehensive approach to ensure integrity and efficient management of a massive number of IIoT for the update process. Based on our approach, we need an integration of different ways to achieve secure communications between IoT devices and software providers, as well as efficient management to register and track the update packages associated with different software components.

Table of Figures

Figure 1:Architecture overview [1] 4
Figure 2: Industrial IoT [4] 6
Figure 2:package dependencies in a typical IoT environment 7
Figure 3: Generic IIoT device layers 8
Figure 4: Sample IoT Devices Dependency Weighted Graph 10
Figure 5- Sample IoT Devices Dependency Weighted Graph. 11
Figure 6:Scoring Equation 12
Figure 7: Sample IoT Device Dependency Chain built based on Sample Data 13
Figure 8:Massive IIoT Software Management Architecture. 14
Figure 9: Software Installation/Upgrade Decision Flow. 16
Figure 10: In-Place Upgrade Mechanism. 17
Figure 11: Rolling Upgrade Devices in Upgrade Domain. 18
Figure 12 Backup Image During Upgrade. 18
Figure 13: Sample Workload Data Details. 19
Figure 14: POC Source Code for Multi-Variate Linear Regression Workload Prediction. 19

References

[1] “Updating IoT devices: challenges and potential approaches.” Accessed: Mar. 25, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9119514

[2] “Rasperry Pi.” Accessed: Mar. 26, 2022. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

[3] “Clear Linux.” Accessed: Mar. 25, 2022. [Online]. Available: https://clearlinux.org/

[4] “Indistrial IIoT picture.” Accessed: Mar. 25, 2022. [Online]. Available: https://www.tibco.com/reference-center/what-is-iiot

[5] “Micro Kernel.” Accessed: Mar. 26, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Microkernel

Dell Technologies believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” DELL TECHNOLOGIES MAKES NO RESPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying and distribution of any Dell Technologies software described in this publication requires an applicable software license.

Copyright © 2023 Dell Inc. or its subsidiaries. All Rights Reserved. Dell Technologies, Dell, EMC, Dell EMC and other trademarks are trademarks of Dell Inc. or its subsidiaries. Other trademarks may be trademarks of their respective owners.