Continuous Delivery of Software on IoT Devices

Diego Prens
Department of Systems and Computing Engineering
Universidad de los Andes
Bogotá, Colombia
da.prens@uniandes.edu.co

Iván Alfonso
Department of Systems and Computing Engineering
Universidad de los Andes
Bogotá, Colombia
id.alfonso@uniandes.edu.co

Kelly Garcés
Department of Systems and Computing Engineering
Universidad de los Andes
Bogotá, Colombia
kj.garces971@uniandes.edu.co

John Guerra-Gomez
Khoury College of Computer Sciences
Northeastern University
California, Estados Unidos
ja.guerrag@uniandes.edu.co

***Abstract*—Given the dynamic environment and changing conditions on the Internet of Things (IoT), developers need to periodically update software and deploy new versions on smart devices and edge devices such as gateways. A software update can generate unforeseen downtimes, or can also alter the device resource consumption. Therefore, we propose an approach that deals with the need for (semi)automating deployment, monitoring and visualization of the impact of software updates on devices operation. We use modeling to abstract the concepts that matter in the domain of continuous software delivery for IoT devices. This abstraction was implemented on top of time series and document-oriented databases.**

***Index Terms*—Continuous integration, visual analytics, containers, deployment**

I. Introduction

The Internet of Things (IoT) refers to many physical devices connected to the Internet around the world that collect and share data [1]. Today, IoT systems have modified and optimized the activities that people perform on a daily basis in various sectors such as health, transport, industry, agriculture, and home. The software embedded in the devices of an IoT system is frequently updated and redeployed by developers. In many cases, this procedure should be done remotely without moving on next to the devices for security reasons or budget constraints. For example, in underground monitoring systems, it is risky and expensive to move to the location of the sensors or actuators to perform a software update. Furthermore, deploying software on multiple devices at the same time is a tough task for IoT Engineers because they have little visibility of low-level details such location or resources consumption of devices.

Each software update that is deployed on a device, could present downtimes that have not been contemplated in previous stages of software development, inducing the unavailability of the system. On the other hand, the average resource consumption of devices such as memory, CPU, energy consumption and bandwidth can also vary among software updates. IoT engineers can use resources consumption data to discover anomalies introduced by new releases. For these reasons, it is important to monitor the operation of IoT devices every time a new release is delivered and compare the releases in terms of resource consumption and faults presented at runtime.

We identified the following requirements in the continuous delivery for IoT devices: *Req1:* IoT engineers should be able to select the devices on which they want to deploy a new release; *Req2:* Deployment of software on IoT devices must be remote and (semi)automated; *Req3:* Resource consumption and downtimes must be monitored for each software version deployed on IoT devices; *Req4:* The IoT engineer should be able to see the monitored data for each software version deployed, through an interactive visual interface.

When comparing our approach to the closest related work [2], [3], one finds that our approach resembles the competition's in three aspects: 1.a) we all take advantage of continuous integration servers, orchestrators, and Docker containers to address Req2; 1.b) Like [3], our approach monitors performance metrics (part of Req3); and 1.c) We took inspiration from [3], to define the different types of deployment (part of Req1). However, our approach distinguishes itself from the rest because: 2.a) Our approach not only monitors performance metrics but also availability (part of Req3); and 2.b) Our approach provides visualizations that any of the studied works have (Req4).

Below we describe the conceptualization of continuous software delivery for IoT devices and how we have instrumented it by using tooling.

II. Modeling and Implementation

Figure 1a) shows the metamodel classes that abstract the concepts of the domain. We have used geometric figures in the upper right corner of each class to explain how the classes relate to the requirements we have identified in the introduction. The abstraction of each requirement in the metamodel is explained below:

A. Req1 and Req2: Deployment Rules and Process

We describe the deployment process and its relationship with the metamodel concepts. Previously, the IoT Engineers (or Contributors) specify the software, the kind of devices (in terms of operating system and resources description) and the devices instances where the software will be deployed. In the metamodel, the concepts Software, Device, DeviceInstance, and Location represent them. After the process is as follow: 1) IoT Engineers commit software changes to a Version Control Server (VCS); 2) The Continuous Integration Server (CIS) automatically makes an executable, test it and build an image from VCS artifacts. 3) The image is stored in an images

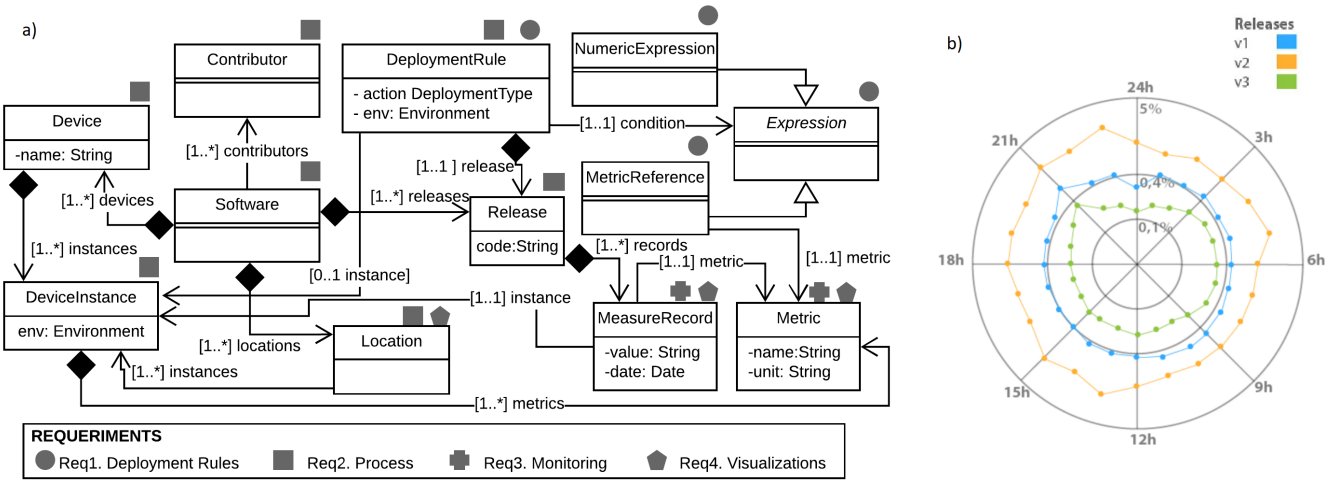

Fig. 1. a) A metamodel for IoT continuous software delivery. For the sake of space, we omitted the enumerations DeploymentType, Environment, RelationalOperator, and OperatingSystem. b) A visualization of CPU consumption of three software releases. For the sake of space, we only show one of the three visualizations

repository; 4) the location and identifier of this commit are persisted in database 5) IoT Engineers sets the deployment rule using a user interface; 6) and its persisted using Orchestrator; 7) IoT Engineer creates a new release from the last commit in the VCS. The DeploymentRule and Release are represented in Figure 1a aand persisted in the database. It is worth noting that the DeploymentRule has a logical expression that relates a Metric and a numerical expected value. A Metric represents performance (i.e., CPU, RAM, bandwidth, CPU temperature) and an availability attribute to monitor. Finally, the rule has an action that it is triggered if the condition evaluation is true. Currently, we have three types of actions: *All* deploys the release on all the device instances associated; *One* deploys the release on a specific device instance; *SpecificProperty* selects the IoT devices which match the values defined, e.g., IoT devices with RAM consumption greater than to 2GB. If the engineer chooses the actions *All* or *SpecificProperty*, he has to specify the environment where the device instances have to be searched; 8) The VCS alerts the CIS about the new release; 9) The CIS sends this information to the Orchestrator which checks if there is any rule associated, in such case the Orchestrator deploys the release on the devices instances that satisfy the rule; 10) Orchestrator sends the deployment request to an M2M communication component using an MQTT Broker; 11) and 12) The device instances catch the request and download and execute a container for the release.

B. Req3: Monitoring

Once the release is deployed, we monitor the operation of the devices as follows: 13) Devices periodically send to the M2M component two stimulus: 13.a) resource consumption data; and 13.b) heartbeats. 14) Resource Monitor component gets this information and stores it in the database. If there is a missing heartbeat, then the monitor will mark the device as down in the database. To make the monitoring possible, the IoT engineer has to guarantee that each device has an Agent component to monitor the resources and send the heartbeats. The concept "MeasureRecord" represents the stimulus sent by the devices.

C. Req4. Visualization

15) Using the previous information, we designed three types of visualizations. The first visualization shows the last release state (devices up or down) and the location of all devices. The second visualization shows the devices historical information about computing resources consumed (Figure 1b). Finally, the third visualization shows historical information about the downtimes of the devices. The second and third ones allow to the IoT engineer to compare historical information of the releases by days or hours.

III. Conclusion and Future Work

In this paper, we propose an approach for the continuous delivery of software in IoT devices addressing needs for deployment, monitoring, and visual analytics. We developed a hardware implementation prototype of a temperature monitoring system to validate our approach. As future work, we plan to: 1) strengthen the validation by applying our approach to a case study; 2) implement machine learning algorithms to (semi)automatically adjust deployment rules in real time; 3) Extend metamodel to address other layer nodes and time-aware concepts.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] J. Bae, C. Kim, and J. Kim, "Automated deployment of smartx iot-cloud services based on continuous integration," in *2016 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2016, pp. 1076–1081.

[3] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, "Foggy: a framework for continuous automated iot application deployment in fog computing," in *2017 IEEE International Conference on AI & Mobile Services (AIMS)*. IEEE, 2017, pp. 38–45.