The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Bachelor Thesis

DEVELOPMENT OF A DEPLOYMENT AGENT FOR THE INTERNET OF THINGS

ENTWICKLUNG EINES DEPLOYMENT AGENTEN FÜR DAS INTERNET DER DINGE

presented by

Philipp Franke

Aachen, March 13, 2018

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Dipl.-Inform. Andreas Steffens

Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Bachelor's thesis entitled

DEVELOPMENT OF A DEPLOYMENT AGENT FOR THE INTERNET OF THINGS

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Official Notification

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

DEVELOPMENT OF A DEPLOYMENT AGENT FOR THE INTERNET OF THINGS

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, March 13, 2018 (Philipp Franke)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, March 13, 2018 (Philipp Franke)

Acknowledgment

First, I would like to thank Prof. Dr. rer. nat. Horst Lichter for the opportunity of writing my bachelor thesis at his chair. I also thank him and Prof. Dr. rer. nat. Bernhard Rumpe for reviewing this thesis.

Special thanks go to my supervisor, Dipl-Inform. Andreas Steffens for his input, continuous feedback and patient.

Furthermore, I thank everybody at gridX for their support specially Joel Hermanns and Robert Fritzsche, who assisted me whenever possible.

Finally, I thank my parents for providing me with constant support during this thesis.

Philipp Franke

Abstract

Over the last decades, continuous integration and deployment (CI/CD) becomes more and more popular within the software development industry. However, applying CI/CD to embedded projects can be challenging. One critical challenge is the volatile environment Internet of Things devices are in. This thesis introduces a deployment agent which is capable of moving the decision problem of doing a deployment to the latest possible point. The deployment agent tackles this challenge by continuously sensing its environment and taking action based on that knowledge. A case study in an industrial context showed encouraging results. Overall, the knowledge base provides a great foundation for further research.

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

> Real programmers don't comment their code. It was hard to write, it should be hard to understand.
>
> ANONYMOUS

Contents

Over the last decades, Continuous Integration and Continuous Deployment (CI/CD) becomes more and more popular within the software development industry. Companies such as Netflix, Google, and Facebook [Rod+17] have adopted Continuous Deployment extensively. Thus, a wide range of companies has utilized these approaches as well, and almost every DevOp knows the phrase "Ship early and ship often" [Ros12].

Through years of evolving the deployment pipelines in a cloud environment, there is also the necessity to adapt these approaches to the embedded software projects. HP Inc. led the way in Continuous Intergration in the context of embedded software development. By implementing continuous integration, they have increased the time spent on writing new features by eightfold [GYF12]. However, testing software on embedded devices is time- and money-consuming because these software products often run on different supported hardware revisions during their lifetime and can often only be tested in an isolated environment. The HP's project manager Gary Gruver says "The problem is, we required over 15,000 hours of testing per day. There aren't enough trees to support this volume of testing" [GYF12].

Although Continuous Intergration is still tricky, Continuous Deployment is even more challenging in regards to the Internet of Things (IoT). For instance, IoT devices can be movable or may have an unreliable network connection[AF+15]. In contrast to deployment pipelines in a well-observable environment where target system is readily available, an IoT device is in a heterogeneous environment where the DevOps might have no control over the devices whatsoever. These environments can lead to slow and even corrupt deployments. Additionally, the number of deployment targets in an IoT scenario [Eva11] will be soon higher than in an ordinary CD scenario, so scalability is a factor as well. One major challenge is the scheduling of deployments on remote devices, especially if a device is not ready for deployments. This does not only mean that a device is not accessible remotely, but a deployment is not appropriate at that time. For example, deploying

software to a driving car is risky as well as deploying trading software where every outage costs money. All in all, the common Continuous Deployment approach cannot cope well with embedded devices regarding weak connections and heterogeneous environment.

Therefore, this thesis tries to solve the mentioned issues by implementing a deployment agent that not only performs the necessary actions to deploy software but also reasons about desired deployments based on the environment.

1.1 Structure of this Thesis

First, chapter 2 introduces the necessary background and defines essential terms. Starting with definition and constraints for the Internet of Things, followed by the clarification of the Deployment Pipeline term and completed by an introduction of reasoning systems.
Afterward, chapter 3 summarizes the problems previously mentioned and distills the challenges concerning the Internet of Things. By using these challenges, it derives requirements which need to be met to tackle the challenges.
Chapter 4 presents the conceptional approach to fulfill the requirements. Hence, it maps the concept to a software architecture. Besides that, it defines the core components in detail.
In chapter 5 the prototype is used to conduct a case study and to evaluate the concepts against the requirements.
Finally, chapter 6 concludes this thesis and proposes potential future work and improvements.

2 Background

Don't panic!

Douglas Adams

Contents

This chapter introduces and explains required foundation: the Internet of Things, Continuous Deployment, Container Technology and Reasoning Systems.

2.1 Internet of Things

IoT has been around for roughly 5 years, but the research community has many definitions for the term. The survey by Atzori et al. from 2010 [AIM10] summarizes the Internet of Things definitions as a convergence of three perspectives:

"Things"-oriented perspective

This perspective concentrates on how to integrate objects or things into a framework. The term, according to the workshop "Beyond RFID - The Internet of Things" organized by the European Commission and EPoSS., can be defined as "Things having identities and virtual personalities operating in smart spaces using intelligent interfaces to connect and communicate within social, environmental, and user contexts" [INF4].

"Internet"-oriented perspective

The "Internet"-oriented perspective focuses on making the object addressable and reachable from any location.

"Semantic"-oriented perspective

With rising numbers of IoT devices, there is also the need for reasoning over generated data and semantic execution environments.

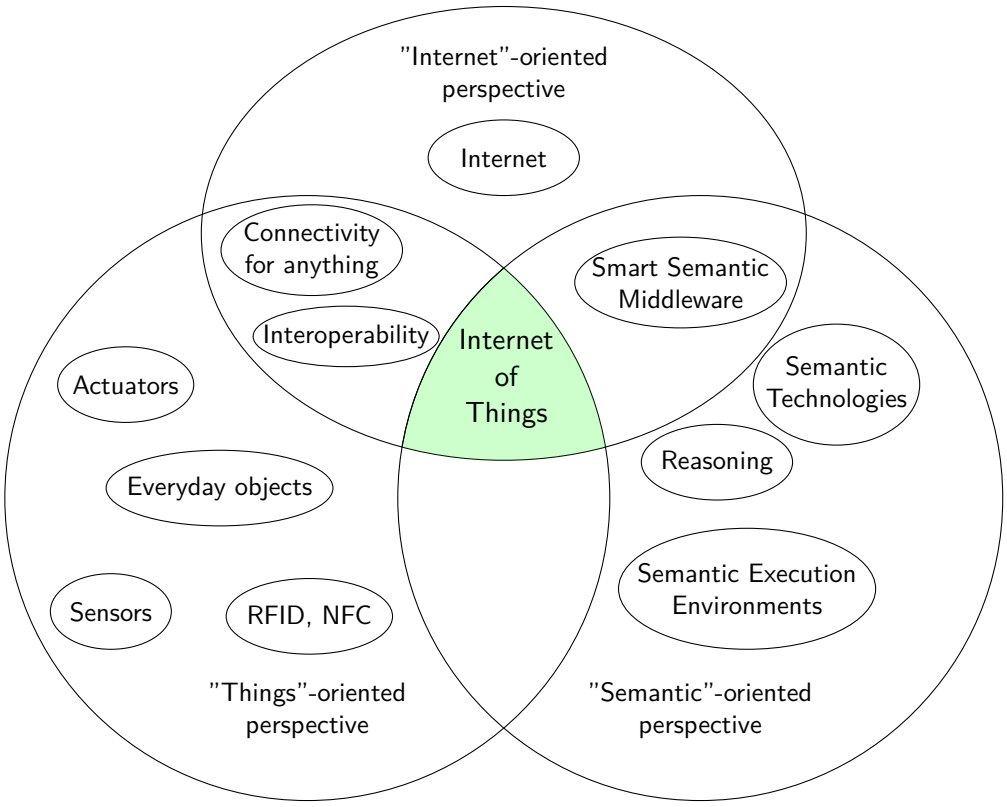


Figure 2.1: Definition of IoT as results of the intersection of perspectives (c.f.[AIM10])

2.1.1 Technical Constraints

The International Telecommunication Union defines IoT as the following: "from anytime, any place connectivity for anyone, we will now have connectivity for anything."[Uni05] Connecting everything with each other anywhere at any time points to the following technical constraints:

Addressing & Networking Issue

It is essential for IoT to bring objects from the physical world into the Internet. Hence, those devices must be addressable by a unique identifier. Technologies like IPv6 and Object Naming Service are ways to tackle this constraint. Nonetheless, the transition from IPv4 to IPv6 is still challenging [Wu+13]. Additionally, the current Internet relies mostly on the TCP protocol for transmission. This, however, is not feasible for IoT because TCP's end-to-end transmission control cannot be used efficiently as stated by Atzori et al.[AIM10]. Furthermore, an IoT device does not even need to sustain its position, nor are all of them connected to the Internet directly. Although there are already advanced techniques like LTE, UMTS, and GPRS to maintain a connection to the internet, they cannot guarantee a reliable connection. Sometimes they are even designed to not have a direct link because of security or privacy issues, so IoT devices yet require a technique to be addressed indirectly.

Hardware Limitations

As IoT also includes low-cost embedded devices, they may be limited regarding energy, computing power, size, and storage. This means that software running on IoT devices has to be optimized for energy efficiency, storage utilization and memory utilization.

Infrastructure

Cisco estimates 50 billion devices which will be connected to the Internet by 2020 [Eva11]. Despite the fact that the estimation includes every manufacturer of such devices worldwide, it is apparent that a manufacturer itself must be able to manage a high amount of IoT devices. Consequently, these manufacturers require an infrastructure which is capable of handling them.

Security & Privacy

Depending on the IoT devices, some of them are collecting sensitive user data which later can be exploited by third parties. Therefore, vendors need to ensure that transmitted data is encrypted and their devices are running the latest firmware. Security and the hardware limitations are overlapping since some cryptographic methods require too many resources which are limited.

2.2 Continuous Deployment

Continuous Deployment bases on Continuous Delivery which is stated in the book Continuous Delivery - Reliable Software Releases Through Build, Test, and Deployment automation by Humble and Farley (see. [HF10]).

In summary, Continuous Delivery is a set of practices (see Section 2.2.1) that aims at building, testing and releasing software quickly, reliably, and frequently with minimal

cost, time and risk. Whereas Continuous Delivery concentrates on the ability to deploy every change to production, Continuous Deployment is deploying it automatically.

2.2.1 Principles

Continuous Deployment mainly relies on the following five principles [HF10]:

Automation The Software Delivery Process should be automated up to the point where specific human decision making is needed. Even though many development teams do things manually because it is easier, they know that the manual approach is more error-prone. Thus, automation is essential for the deployment pipeline.

Build quality in Quality needs to be built into the product in the first place. In other terms, techniques like continuous integration and comprehensive automated testing are designed to catch defects as quickly as possible. The concept behind that was adopted from W. Edwards Deming who was one of the pioneers of the lean movement.

Small Changes By keeping the number of changes small and the time between two releases short, the risk associated with releasing decreases.

Everybody is responsible for the Delivery Process A fundamental requirement is to get everybody committed to the software delivery process so that the whole team can release valuable software faster and more reliably.

Continuous Improvement During the evolution of an application, the delivery process evolves, too. It is crucial that entire team is involved in the development of the delivery process.

2.2.2 Deployment Pipeline

The deployment pipeline a necessary part of Continuous Delivery. Along with [HF10], it models the delivery process of a software product. In another term, it represents the procedures for getting software from version control into the customer's hands.

2.3 Container Technology

The Container technology is in the broadest sense around since 1979. Since then, chroot has been include in Unix distributions [Bib]. Chroot is a tool which changes the apparent root directory of the current running processes. Consequently, a program in such environment cannot access files outside the selected directory tree.

This concept has been improved over last few years, and with the introduction of the kernel namespace feature, Linux systems have been able to create separate instances of namespaces (c.f.[**namespaces(7) - linux manual page**]). Linux namespaces are

called filesystem, PID, network, user, IPC, and hostname. For instance, PID namespaces isolate the process ID space, which means processes in different PID namespaces can have the same process ID. Moreover, the Linux control groups subsystem is responsible for grouping processes and managing their resource consumption (c.f. [MJL08]). Henceforth, containers can be isolated from other containers, and they can have resource restrictions, too.

In other words, a container is a set of processes which are isolated from the rest of the system, running from a distinct filesystem providing all necessary files for running these processes.

2.3.1 Benefits

Using container technology provides various benefits:

Platform Independence The primary benefit of containers is their portability. An application and its corresponding dependencies can all be part of the container. This enables users to run applications in different environments like local desktop, servers or embedded devices. For instance, since the Docker container engine has been ported to various CPU specific architectures, users can start containers on several ARM variants or POWER architectures (c.f.[Est17]).

Resource Efficiency Containers have nearly no overhead and do not require a hypervisor so a single host can have more running containers than virtual machines (c.f [Fel+15]). However, the underlying container engine introduces some bottlenecks due to the tradeoff between ease of use and performance.

Isolation Containers are running in an isolated environment, which means that processes running within a container are not able to affect processes in another container, or in the host system. (c.f. [Inc18])

Scalability Due to small launch times of containers, DevOps can quickly launch them and even shut them down on demand. Furthermore, scaling horizontally is more manageable as containers are designed to be duplicated.

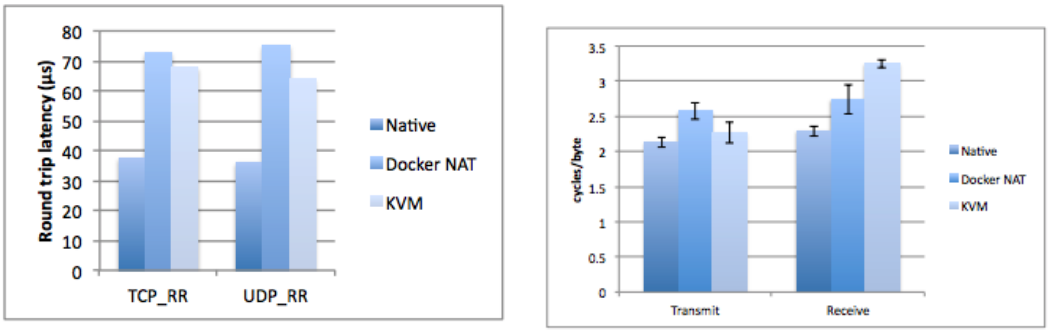
2.3.2 Performance

The previous section outlines the benefits of containers, but these benefits come at a risk that the container engine consumes too many resources. It is indisputable that the native execution of binaries is more performant than executing them inside the container or virtual machine. Nevertheless, it is important to understand whether containers are suitable for the Internet of Things concerning performance.

The IBM Research Report "An Updated Performance Comparison of Virtual Machines and Linux Containers" by Wes Felter at el. is comparing native, non-virtualized execution to executions within a container. In order to determine the overhead, they have measured

workload metrics such as throughput and latency (c.f. cite[Fel+15]).They choose Docker as representative container manager which uses containerd as container engine (c.f. [Hyk17]).

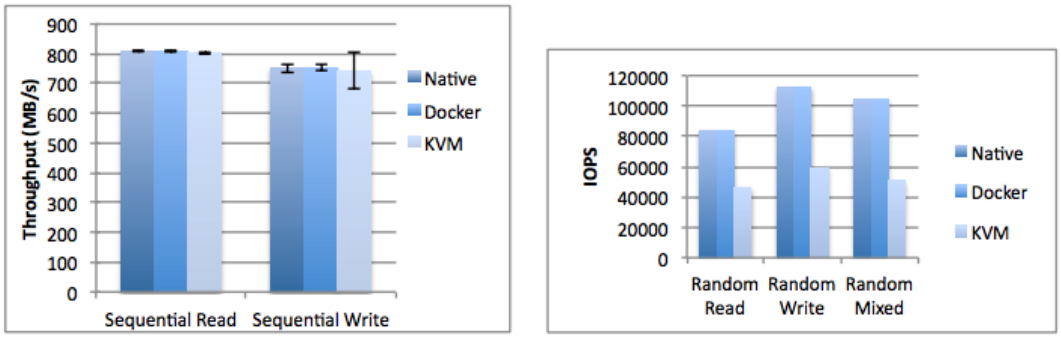
The following figures summarize the findings of the IBM Research team:



(a) Network round-trip latency (μs) (b) TCP bulk transfer efficiency (CPU cycles/byte)

Figure 2.2: Native/Container Network Benchmarks [Fel+15]

Figure 2.2a shows the transaction latency measured by netperf for TCP and UDP. Docker's NAT doubles the latency for both protocols compared to the native execution. As shown in figure 2.2b, Docker, especially Docker's NAT, needs more CPU cycles per byte in both directions. Hence, the identified CPU overhead reduces the general performance for network-intensive tasks.



(a) Sequential I/O throughput (MB/s) (b) Random I/O throughput (IOPS).

Figure 2.3: Native/Container Block I/O Benchmarks [Fel+15]

Figure 2.3a shows sequential read and write performance averaged over 60 seconds. Clearly, there is almost no overhead. Furthermore, figure 2.3b presents the performance of random read and write and there is still no overhead.

In conclusion, Docker introduces negligible overhead for CPU and memory usage. However, for I/O-intensive task container managers may be the bottleneck if the configuration

is not tuned. Because the overhead added by containers is imperceptible, IoT devices are able to run containers.

2.3.3 Security

As mentioned above, container engines use kernel features such as kernel namespace to isolate process so that processes within a container cannot affect processes running elsewhere. Besides that, each container has its own network stack, so they are not able to get access directly to the sockets or network interfaces.
Control groups add an extra layer of safety using the ability to limit resources. By means of the kernel feature Capabilities, containers can have a more fine-grained access control system which means that they can run with a reduced capability set. For example, it is possible to deny access to perform I/O port operations.
Finally, there are already known systems such as AppArmor, SELinux, and GRSEC which can also be used to harden a host of containers.

2.4 Reasoning systems

A reasoning system is a software system which cames to conclusions based on available knowledge by means of logical techniques. Two of these techniques are:

Deduction Deduction is a method of reasoning from premises to reach a logically certain conclusion (see figure 2.4).

Induction Induction is the process of reasoning in which the premises are seen as strong evidence for the truth of the conclusion (see figure 2.4).

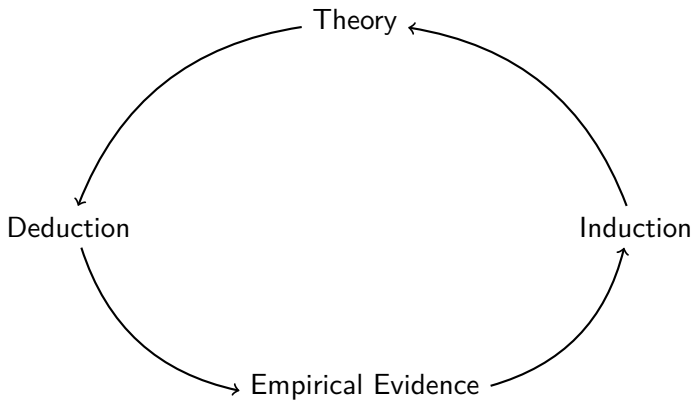


Figure 2.4: Connection between Deduction and Induction

2.4.1 Types of reasoning system

The following types of reasoning systems are overlapping to a certain degree and this list only includes reasoning systems that are relevant to this thesis.

Rule engines Rule engines primarily have a set of rules where each rule has two parts: the condition and the action. A rule engine determines when it evaluates the rules and triggers the action if the condition matches the current state of the world. That is a well-known implementation of a rule engine called Production Rule System.

Procedural reasoning systems Procedural Reasoning System (PRS) is a framework for building real-time reasoning systems which are able to execute complex tasks in changing environments. It relies on the procedural knowledge representation describing how to complete a set of actions to reach a goal. PRS might only have partly specified plan for the future. Furthermore, this system is based on belief-desire-intention-framework (BDI) for intelligent agents. To clarify the connection between both frameworks: PSR is reasoning over facts ("beliefs") to choose suitable plans ("intentions") for a given goals ("desire").

2.4.2 Knowledge-based System

A knowledge-based system is usually organized into two subsystems: a knowledge base, and an inference engine.

Knowledge Base A knowledge base can store complex structured and unstructured information which are provided by applications. The term was used to distinguish this kind of knowledge store from the widely used term database because the requirements for the first knowledge-based systems were the opposite of those of databases at that time (the 1970s). Another key point is that the object model with classes, subclasses, and instances are the exemplary data structure for a knowledge base because object models can describe complicated facts.

Inference Engine The inference engine is subsystem which applies logical rules to the knowledge base to induce new knowledge.

2.5 Summary

This chapter introduced the required background to understand this thesis. It, first, explained the Internet of Things and presented the related technical constraints. These constraints will assist as challenges through this thesis. Second, the chapter provided the necessary Continous Delivery principles and defined the associated terms. Additionally, container technology was explained, and their benefits were outlined. Ultimately, it introduced relevant reasoning systems, especially rule engines. The next chapter describes the problems, challenges and the requirements of a deployment agent.

3 Problem Statement

There are no facts,
only interpretations.

Friedrich Nietzsche

Contents

After a brief introduction to the thesis in chapter 1 and a discussion about the background in chapter 2 , this chapter focuses on the problem this thesis tackles. First, we identify the main challenges regarding deployment pipelines for the Internet Of Things; then we deduce the requirements from the recognized challenges.

3.1 Challenges

As mentioned in the previous chapters IoT devices have specific technical constraints. The technical issues classified by Hanoon et al. [Har+16] lead to the following problems:

P1 - Connectivity "[T]he reliable connectivity demands to address and sense the devices all the time" [Har+16]

P2 - Hardware Limitiation "The hardware issue with power and storage constraint also required to be managed" [Har+16]

P3 - Scalability "To address huge number of [..] nodes in a world-wide network is also referred as major scalability issue" [Har+16]

P4 - Heterogeneous Environment "Due to their size and frequently changing location property devices are not able to access the power all the time" [Har+16]

In essence, these problems exist due to the needed mobility of an IoT device. Furthermore, the problem connectivity is also mainly influenced by the rapid changing or unknown target environment. Therefore, the problems above reveal four significant challenges for deploying on the Internet of Things devices:

C1 - Context-awareness A deployment system needs to be aware of its environment to operate correctly. If an IoT device is movable, the agent should be aware of it. The context can be distinguished between the following categories

C1.1 - Environment According to the collected sensor information, a deployment agent has to minimize amount of artifacts which needs to be downloaded.

C1.2 -Rapidly Chaging environment A deployment agent has to cope with the current user activity. For example, a car's deployment agent should not perform a deployment while it is in a deployable state.

C2 - Fault-tolerance An IoT device is often not physically accessible; therefore a deployment agent has to be able to always reach a stable deployment state.

C2.1 - Atomic deployment A deployment agent should minimize the risk for unsuccessful deployment and should rollback to stable deployment if necessary.

C2.2 - Fast rollback The deployment agent should be designed to fail fast if some unexpected condition is reached.

C3 - Scalability From a release engineer perspective, the deployment pipeline needs to handle many different devices in different versions and states.

C3.1 - Usability The deployment pipeline should minimize the amount of time to perform an deployment on various IoT.

3.2 Requirements

By following the challenges above, we can identify relevant user stories which help us to extract the essential requirements.

User Stories

US-1 As a DevOp, the deployment should not set in motion if there is not enough disk space left on the device.

US-2 As a deployment agent, I want to download the artifact before a deployment.

US-3 As a DevOp, I want to be sure that a deployment is not set in motion if the CPU is under load (>70%).

US-4 As a DevOp, I want to be sure that a deployment is not set in motion if the memory usage is high (>80%) so that necessary system components have enough memory.

US-5 As a DevOp, I want to be able to force a deployment.

US-6 As a DevOp, I want to have stable last working image on the device to easily rollback.

US-7 As a DevOp, I want to be sure that the deployment agent restarts applications with a new configuration if the configuration has changed.

US-8 As a DevOp, I do not want that the deployment agent kills/stops manually started processes.

US-9 As a IoT developer, I want that the deployment agent has small memory, CPU and disk footprint.

US-10 As a DevOp, the deployment should not set in motion if the device is actively used by the user.

US-11 As a DevOp, I want to trigger deployment on a subset of devices, and the deployment should be in a declarative way.

US-12 As a DevOp, I want to know which deployments were scheduled/pending/running/successful/unsuccessful.

US-13 As a DevOp, I want the agent to receive a syntactically correct and complete deployment configuration.

Essential Requirements

In line with the user stories and challenges, we can state the following requirements that the deployment agent should meet.

Req-1: Sensing of Environment The deployment agent should support a way to sensing the environment. Therefore, the agent should have sensors that keep track of the device internals and should also be extendable by external sensors. (C1 - Context-awareness)

Req-2: Inference Ability Inference Ability is a major requirement so that the deployment agent can make sense of the collected sensor information. This ability should allow the deployment agent to deduce new knowledge. (C1 - Context-awareness)

Req-3: Action Selection Since the agent is in a potentially uncontrollable environment, it needs to support action selection. In detail, it needs to select the appropriate action based on the inferred knowledge. (C1 - Context-awareness , C2.2 - Fast rollback)

Req-4: Reliable Deployment State The deployment agent has to maintain a well-defined deployment state at all time. This minimizes the risk of leaving devices in an uncontrollable state. (C2.1 - Atomic deployment C2.2 - Fast rollback)

Req-5: Flexibility Due to the challenge C3 - Scalability , the agent has to support easy integration of different devices in different environments. These environments might require different kinds of sensors, different types of data transport mechanisms and also diverse sets of inference engines. (C3 - Scalability)

3.3 Summary

This chapter introduced several challenges regarding the deployment process for the Internet of Things devices. We identified three challenges based on the problems stated by Hanoon et al.: The first challenge is context-awareness. As IoT devices are in heterogeneous environments, a deployment agent needs to cope with these environments. The second challenge we recognized is fault-tolerance. Deployment pipelines are used in a well-known and readily accessible cloud environments, whereas the last mile of an Internet of Things Deployment Pipeline usually ends in an uncontrollable environment. The last challenge is about scalability. After specifying the requirements, we can design the concept in the next chapter.

4 Architecture & Implementation

> It's not a bug - it's an undocumented feature.
>
> Author Unknown

Contents

After setting the requirements for the deployment agent, it is important to design an appropriate concept which will meet the requirements specified above. This chapter presents the architectural overview and then dives deeper into the several components.

In a nutshell, the deployment agent is responsible for reliably deploying artifacts into volatile environments by sensing its environment and inferring possible problems to minimize the risk of unsuccessful deployments.

4.1 Architectural Overview

Our goal is to have a clean separation of concerns. Therefore, the deployment agent is divided into five types of systems: process manager, knowledge base, rule engine/orchestrator, sensors, and actors.

In the following section, responsibilities of the related systems are briefly described:

Process Manager The process manager is the interface to communicate with the supervised processes. It is responsible for keeping track of all running supervised processes and creating new ones. A supervised process represents an arbitrary long-running task, which can be started, stopped and monitored. This system also takes care of maintaining a constant deployment state by reconciling the state of all running processes and all desired-to-run processes.

Knowledge Base As described in chapter 2, the knowledge base is responsible for storing facts. In this case, the facts are internal system information such as memory usage and CPU utilization and knowledge such as deployments and authentication tokens. The inference engine is also part of the knowledge base, and it is able to deduce new facts from existing ones.

Rule Engine/ Orchestrator The rule engine is the hearth of the system. It maintains a set of rules and their corresponding actions and continuously reevaluates the current state of the environment against the rule set. If a condition of the rule set is fulfilled, the rule engine triggers the related action.

Sensors There are two types of sensors: internal and external sensors. The internal sensors are collecting system information such as CPU usage, memory utilization and network throughput. In contrast, the external sensors are responsible for domain-specific facts. For example, external sensors could be collecting deployment information, the availability of artifacts or even information exposed by processes.

Actors Actors are receiving input and locally decide what to do next. In fact, they can perform a specific task and send new messages to other actors. In our case, they are doing the actual task.

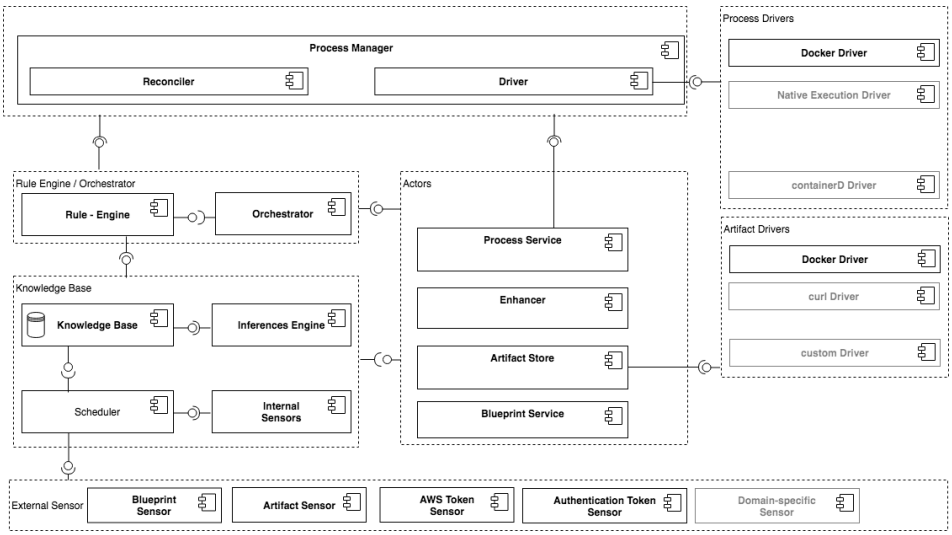


Figure 4.1: Architectural Overview

4.2 Core Components

In this section, we define the task of each subsystem and its related components. It also covers the intention behind it.

4.2.1 Process Manager

The following introduces the two components from the process manager, as seen in figure 4.1.

Drivers

In order to have a flexible system that is capable of managing different kinds of processes such as native processes and isolated process like containers, we consider an approach inspired by device drivers which provide a software interface to hardware devices.
Thus, the driver abstracts away the process management. In detail, process manager uses the driver to list its running processes and retrieve status information about a particular process.

```
1 // Manager is the interface which allows to list and transform
     process
2 type Manager interface {
3     // Get returns a particular supervised process.
4     Get(ctx context.Context, pid string) (Process, error)
5     // List returns all supervised processes.
6     List(context.Context) ([]Process, error)
7     // Transform transforms a blueprint into a process.
8     Transform(context.Context, interface{}) (Process, error)
9 }
```

Source Code 4.1: Process Manager

Moreover, the process manager needs to be able to create a new process. Therefore, the driver also needs an interface for that. This means in effect, that the process manager can manage all kinds of processes as long as there is such a driver. For instance, if there is an implementation for native execution, the process manager can easily handle them due to that abstraction. At this point, the implementation is not relevant because it is part of the next chapter. Because of the abstraction, it makes sense to abstract the process itself so that the process manager does not need to know how to start or kill a process.

```
// Process represents a process managed by a process manager
type Process interface {
    // ID is an unique identifier for a process.
    ID() string
    // Driver is the identifier for the driver.
    Driver() string
    // Start starts a process.
    Start(context.Context) error
    // Status returns the process' current status
    Status(context.Context) (Status, error)
    // Kill kills/stops the process.
    Kill(context.Context, syscall.Signal) error
}
```

Source Code 4.2: Process Specification

All in all, the driver approach allows the process manager focus on the behavior and not how to do it. Hence, the process manager gives us the flexibility to run any process on IoT devices. For example, a device which does not support containers due to hardware limitation can still use a native execution driver.

Reconciler

The other component of the process manager is the reconciler. Its purview is the minimization of the amount of required deployment action such as starting and stopping processes and maintaining a valid deployment state.
The reconciler basically tries to reach the desired process list from the current process list. In other words, the reconciler starts all desired processes and stops rest.

```
for {
  desired := getDesiredProcessList()
  current := getCurrentProcessList()
  makeChanges(desired, current)
}
```

Source Code 4.3: Simplest implementation of the reconciler

Although this works, it is not the most efficient way and not even a useful one because an added process should not interfere with an already running process. Consequently, the reconciler finds the intersection of both process lists, so it only needs to stop/start the difference of the intersection and the current process list/desired process list, respectively. In figure 4.2, there are two running processes(P1, P2) and two processes(P2, P3) which are desired to run. In this case, the reconciler stops P1 and starts P3.
Moreover, the reconciler is responsible for maintaining a valid deployment state. If the process service calls the reconciler, it guarantees the running process list after reconciling is either the desired process list or the previous desired list. For Instance, if a process cannot be stopped/started during the reconciliation, the reconciler automatically rolls

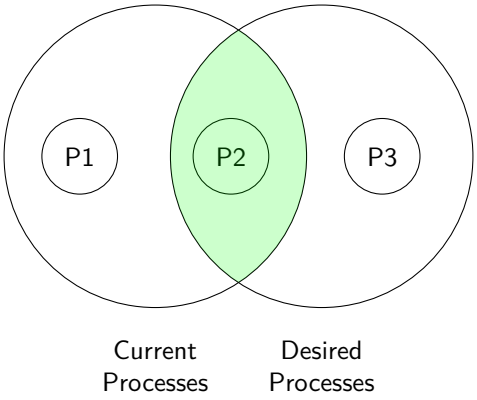


Figure 4.2: Intersection of process lists

back to the previous desired process list. This design decision is made with the assumption that the previous desired list is quality-assured.
The reconciler tries to fulfill the requirement of Req-4: Reliable Deployment State as mentioned in chapter 3.

In conclusion, the process manager as the name suggested manages processes. However, it is not important for the process manager what kind of process it is. It only matters that a process can be started, stopped or retrieved its status. The decision design allows us to strictly separated the concerns. It also meets our requirement Req-5: Flexibility.

4.2.2 Knowledge Base

The subsystem knowledge base is a combination of some data store which saves facts and the inference engine. In the following, the design of the database is discussed and why it was chosen. This section also specifies the capabilities of the inference engine and ends with the defining the role of the scheduler.

Fact

Since the knowledge base is about facts, we need to define the term first: A fact identified by a key is arbitrary information at a specific time.

```go
// Fact represents an information about the environment which is
// available through the knowledge base.
type Fact struct {
    // Namespace is represents a logical group of facts.
    //
    // This property is required and cannot be change after
       creation.
    Namespace string

    // Key is a unique identifier within a given namespace.
    //
    // This property is required and cannot be change after
       creation.
    Key string

    // Value should hold the information used by the rule engine
       to make a
    // decision.
    //
    // This property is required but mutable.
    Value interface{}

    // CreatedAt is the time at which the fact was created.
    CreatedAt time.Time

    // UpdatedAt is the time at which the fact was updated.
    UpdatedAt time.Time

    // ExpiresAt is the time at which the fact is no longer
       available.
    ExpiresAt time.Time

    // Owner is the writer which inserted fact.
    //
    // This property is required and cannot be changed after
       creation.
    Owner string
}
```

Source Code 4.4: Fact Specification

Store

In principle, the store is a database which stores facts. In our case where the deployment agent is in a rapidly changing environment, the store has special requirements.
First of all, the store has to be able to forget facts. This ability is necessary due to the environment of IoT devices. For example, when the deployment agent of a mobile device stores the current location in the knowledge base while the device is moving, the stored

fact might be invalid soon. Thus, facts need an expiry date so that the knowledge base can also represent volatile facts.
Another reason for this feature is that IoT device usually have limited amounts of disk space, and in that case, the knowledge base automatically frees some of its taken spaces. Nonetheless, the knowledge base is able to restore previous versions of facts. This feature should only be used on powerful devices.
Furthermore, the store has to provide some security mechanism. Otherwise, a sensor could override someone else's fact so we decide that every fact an owner which is the only one who can override an already present value.
Another feature the store needs to support is transactions because they are necessary to get/save multiple related facts at once securely. Otherwise, the stored data might be inconsistent.
This store is the foundation for tackling the requirements Req-2: Inference Ability and Req-1: Sensing of Environment.

Scheduler

Another component of the knowledge base is the scheduler. In general, the scheduler is responsible for periodically retrieving facts from the registered sensors and stores them into the store.

A sensor described in detail in next section is collecting data about the environment or system internals and transforms them into a set of facts. Because almost all sensors have in common that they need to sense frequently due to the changing surroundings, the deployment agent manages these sensors by means of a scheduler. We decide to decouple the sensors from the knowledge base because in this case, the sensors can primarily focus on sensing data. A sensor only needs to register itself at the scheduler and provide two methods: the first method returns the time interval and the second method returns a set of facts which should be atomically stored. This abstraction allows the sensor dynamically to adjust its time interval which is useful for information with a fixed expiry date such as authentication tokens. The scheduler also has other advantages like different scheduling algorithms, sensor timeouts. In most cases, the First-In-First-Out algorithm is suitable until long-running sensor are registered.

The scheduler also contributes to meet the requirement Req-2: Inference Ability.

Inference Engine

The inference engine is the last component of the knowledge base. In a nutshell, the inference engine applies a set of conditional rules to facts from our store to deduce new facts.

The following approach is different to a conventional inference engine because it does not infer over the entire knowledge base. In this approach, the engine has a set of conditional rules which are applied to a specific fact. After all possible facts have been deduced, the engine provides a list with the inferred facts. For instance, the inference engine gets a docker blueprint as input. It then starts looking into the knowledge base

to determine if the docker image has been already downloaded. Therefore, it uses the two isolated facts the docker blueprint and docker artifact to extract the new fact that the image is available. This fact is later used by the rule engine to decide whether the deployment process can continue.

4.2.3 Rule Engine/Orchestrator

Due to the flexibility of the provided architecture, the deployment agent needs a method to handle arbitrary tasks. There is a rule engine which evaluates conditions based on the facts of the knowledge base and the inferred ones. These conditions are triggering their corresponding actions which are handled by the orchestrator. The orchestrator assigned the different task to the appropriate actor who tries to perform the task. After the actor is done, it reports the results back to the orchestrator so the orchestrator can assign a new task.

In figure figure 4.3, there is a typical successful deployment flow which illustrations how we achieve that successful deployment. In the beginning, the blueprint sensor polls some arbitrary remote server for blueprints and stores the response in the knowledge base. Secondly, the rule engine recognizes that "new blueprint available" condition is valid and triggers the transformation from an external blueprint into an internal one. This same procedure repeats a few times until the blueprints are transformed into processes. At that point, the reconciler checks if the subset of processes has the permission to get deployed. If so, we start reconciling and report the result back.

The rule engine and the orchestrator are also meeting our requirement Req-3: Action Selection.

4.2.4 Sensors & Actors

This section presents the last two subsystems: actors and sensors. At first, we describe sensors and actors in general, and then we dive deeper into the important sensors and actors.

Sensor

A sensor has an interface to collect information data, and it transforms the collected data into a fact. At this point, it does not matter what the fact semantically means because the rule engine tries to make sense of it. Nevertheless, it is important to choose the data type of a fact carefully. Otherwise, the rule engine might not be able to evaluate a condition.

In particular, if a sensor collects temperature data and stores it as a string, the rule engine cannot report whether the temperature is under a specific threshold because it could not cast the string to a number. Due to the fact that the knowledge base is designed to be generic, it is up to the rule engine or actor to make sense of a fact.

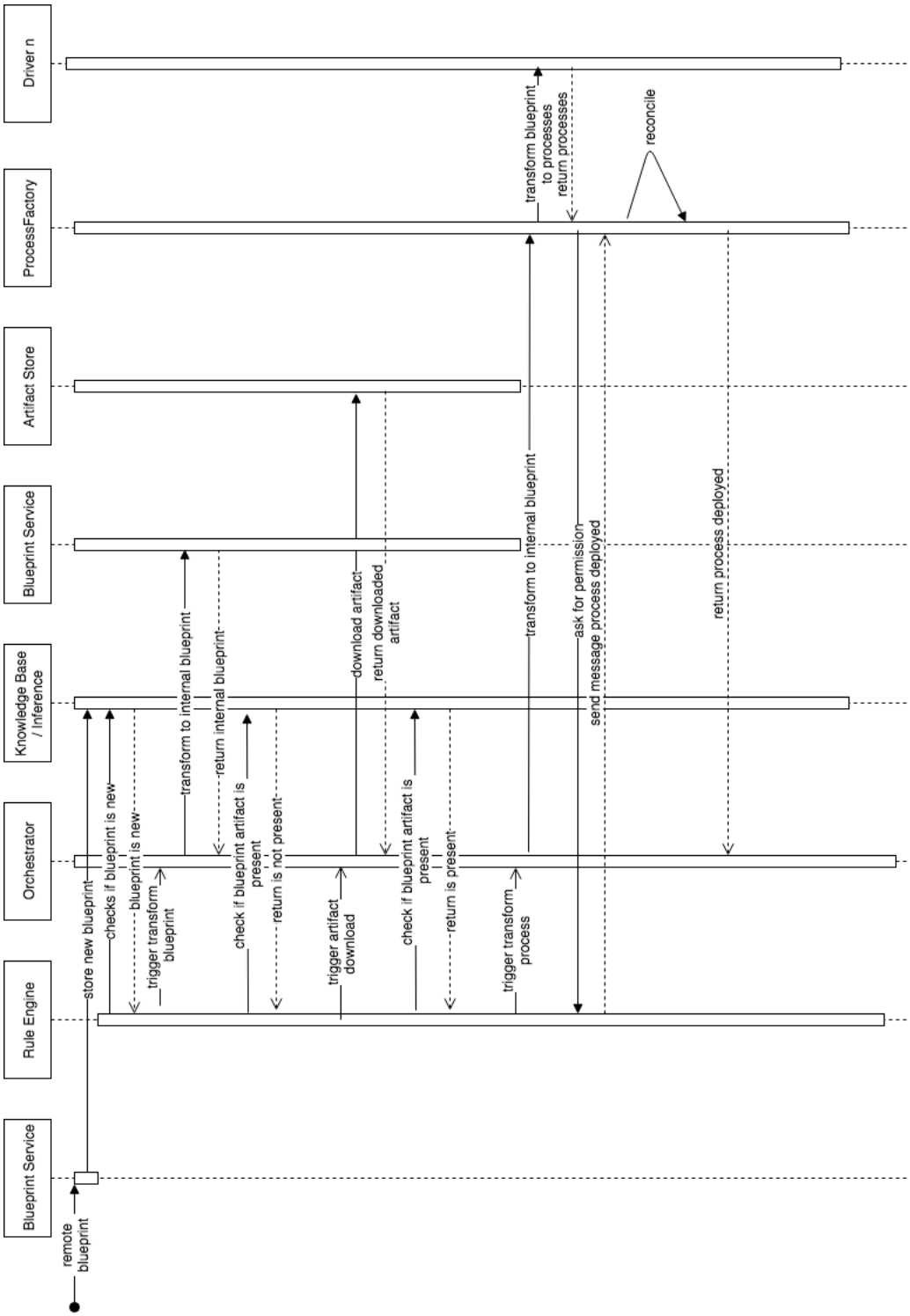


Figure 4.3: Sequence Diagram: Successful deployment

```go
// Sensor is interface a sensor needs to implement.
type Sensor interface {
    // Name represents the name of a sensor and
    // is also used as owner.
    Name() string

    // Interval specifies time interval between two runs.
    Interval() time.Duration

    // Run returns a list of facts which can be stored in
    // the knowledge base.
    Run(context.Context) ([]*knowledge.Fact, error)
}
```

Source Code 4.5: Sensor Specification

Internal Sensors The internal sensors are collecting system internals such as CPU, memory and disk usage.

Blueprint Sensor A central sensor is the blueprint sensor. Its responsibility is to collect deployment blueprints from an external server. This sensor usually polls a remote server. Nevertheless, it also possible to provide an API interface to push blueprints.

Artifact Sensor This sensor collects information about the present artifact in the artifact store.

Actor

Actors are the part where the actual task is performed. In detail, the orchestrator has a task which needs to be done; therefore, the task is forwarded to a specific actor. While the actor is performing its received tasks, the orchestrator is already sending tasks again. If the task is done, the actor reports back to the orchestrator and is ready for new tasks.

Blueprint Service The Blueprint Service actor is responsible for transforming an external blueprint into an internal blueprint which can be used by a different actor.

```go
// Blueprint is a template for a process.
type Blueprint interface {
    // Driver is the driver used to handle the blueprint.
    Driver() string
    // Blueprint is the actual blueprint.
    Blueprint() interface{}
}
```

Source Code 4.6: Blueprint Interface Specification

```
// Blueprint represents a template for docker container.
type Blueprint struct {
    Image string

    // Process
    Entrypoint []string
    Cmd        []string
    WorkingDir string
    Env        []string

    // Behaviour
    RestartPolicyName       string
    RestartPolicyRetryCount int
    AutoRemove              bool

    // Restrictions
    User       string
    ULimits    []ULimit
    Privileged bool

    // Mounts
    Mounts []Mount

    // Network
    Hostname     string
    NetworkMode  string
    PortBindings []PortBinding

    // Annotations
    Labels []string
}
```

Source Code 4.7: Example of an Internal Docker Blueprint

Process Service The process service has two important task. First, the process service transforms by means of the process manager a blueprint into an actual process. Although this allows the service to start processes, it uses the reconciler for a robust deployment. Before the service triggers the reconciler, it ask the rule engine if its allowed to deploy.

Enhancer Service The Enhancer Service takes as input device-specific blueprint and has the ability to modify the blueprint by adding information from the knowledge base. In detail, this means that a DevOp can set a variable named like a key in our knowledge base in blueprint configuration and the enhancer will replaces it.

Artifact Store The artifact store takes as input string, which is resolved to a docker registry repository or to URL. With this information the artifact store is able to download the needed artifacts.

We choose this approach in order to have the flexibility to add/remove functionality based on devices the deployment agent is running on.

4.3 Technology

The deployment agent is entirely written in Go[Gol]. This was a design decision because of the following benefits:

Cross-Compiling The Go's compiler allows to cross-compile the source for different architecture or operating system. In case of this thesis, the agent need to be compiled for arm.

Concurrency Go support native concurrency, which makes it easy to use. As the deployment agent needs to handle multiple tasks at the same time, it makes go a good fit.

The knowledge base is based on badger [Dgr], which is a key-value store originally built for dgraph. Benefits of badger are that it is optimized for solid state disk. Badger also support TTL key, which expires after a particular time.

Docker[Doc] is the prefered container engine at time of this thesis, because it provides an easy to use api and enough libraries to interact with it.

4.4 Summary

In essence, we presented our architecture and how we are going to meet the defined requirements. We introduced the process manager that is responsible for managing supervised processes. Furthermore, the knowledge base was specified as the brain of the entire deployment agent. Then, the orchestrator and the rule engines have been stated. They manage the incoming tasks. Moreover, we have determined actors and sensors in our concept. The architecture covers at least one requirement so we can move on to the next chapter where the prototype is implemented.

5 Evalutation

> If debugging is the process of removing bugs, then programming must be the process of putting them in.
>
> Edsger Dijkstra

Contents

The evaluation is an important part of this thesis. Therefore, we conducted a case study to gain some insights. This chapter ends with a discussion about the architecture and some design decision.

5.1 Case Study

A case study is exploratory research method. The central goal is to understand what is happening and to seek new insights which might lead to further research. The following section presents our case study:

5.1.1 Context

We conducted the case study in an industrial context at gridX GmbH[1]. GridX is a German energy startup with approximately 20 employees. They are selling various products such energy tariffs, smart meters, and gridBoxes. Their central goal is to create a peer-to-peer energy network. In other words, they are trying to become the largest energy supplier without own power plants by utilizing photovoltaic systems and battery storages.

[1] www.gridx.de

The product which is relevant for our case study is the gridBox. The gridBox is an IoT device, that is monitoring inverters, battery storage, energy meters and even electric vehicle charging stations. It is also able to control inverters and optimize the self-consumption rate. gridX has already shipped 180 gridBox since last year. The gridBox is running a custom made operating system based on linux which runs a minimal base system while all applications are running in docker containers on top. That makes this IoT device suitable for our case study.

Deployment Pipeline

The deployment process consists of two stages: the build stage and the deployment stage. In the build stage, the applications are built and tested and published in an artifact repository. The deployment stage is separated into 4 steps:

1. Ask someone with remote access to a gridBox to deploy a new version.
2. Person generates a new authenticaton token to access the artifact repository.
3. Person modifies the update script to deploy the latest version.
4. Person executes the update script for each device.
   a) Script stops old containers.
   b) Script pulls docker images.
   c) Script starts new containers.

5.1.2 Objectives

Since the previous section defined the case study's context, we need to specify our objectives. Each objective is summarized and also features which data we are required to collect to answer the objectives.

Impact on Deployment Robustness

Description A key challenge is C1 - Context-awareness which the deployment agent tries to tackle. Another important challenge is C2 - Fault-tolerance which is tightly coupled to the requirement Req-4: Reliable Deployment State . Thus, we need to explore the impact of deployment on that matter.

Required Data In order to indicate whether the deployment process is robust, we have to perform broken deployments. We measure the time until broken deployment was resolved.

Impact of Scalability

Description The third challenge was C3 - Scalability which is also linked to the Req-5: Flexibility . Because of the importance, we want to explore if the requirement is fulfilled.

Required Data To see the impact of the scalability, we need to deploy two several boxes at once. By measuring the duration, we are able to judge if our approach is scalable.

5.1.3 Collected Data

As conducting the case study, we collected the following data:

Validation of Deployment Blueprint

Approach To gain evidence about the robustness, we intentionally deployed a broken container specification. And we measured the time until the issue was resolved.

Value Immediately and the previous process list was running without an interruption

Related Objective Impact on Deployment Robustness

Multiple of Deployments at Once

Approach To gather information about our scalability, we need to be able to perform multiple deployments. We select a group of ten gridboxes for this scenario.

Value 10/10 boxes has been running the latest version within 4min.

Related Objective Impact of Scalability

Failure of Preconditions

Approach To gather information about our failure of preconditions, we need to be able to try deployment, when precondition is not valid.

Value yes, it prevents deployment when preconditions are not fulfilled.

Related Objective Impact of Scalability

5.1.4 Conclusion

The case study was conducted at gridX to judge if our deployment agent tackles the requirements. In particular, our case study was motivated by three objectives detailed in the section 5.1.2.

Impact on Deployment Robustness At mention above, the previous deployment process stops the containers and then pulls the new images. Thus, there was not a robust deployment pipeline because it was manually handled.
Furthermore, indicates that deployment agent introduces a robust way to deploy onto embedded devices because it was able to prevent broken deployment at an early stage.

Impact of Scalability Since the previous deployment process was done sequentially, the deployment agent is able threefold the speed of deployments by polling the server instead of actively sending updates to the gridBox.

In conclusion, the deployment agent was able to make the deployment pipeline more robust and even more scalable.

5.2 Discussion

5.2.1 Req-1: Sensing of Environment

Summary The deployment agent is equipped with sensors which is sensitive to its surroundings. For example, there is a disk sensor which periodically collects disk metrics. Metrics are continuously stored in the knowledge base.

Evaluation In general, it is important to have sensors like CPU utilization, memory usage, and disk usage. Furthermore, it also good to have custom sensors which values can later be read by the different actors. The approach managing the time interval for a sensor with a scheduler might be not necessary. However, the abstractions make easy to integrate new sensors.

5.2.2 Req-2: Inference Ability

Summary The inference engine of the deployment agent can deduced new facts from existing knowledge. Based on the new deduced fact, the rule engine signals the orchestrator to dispatch actions.

Evaluation Having an inference engine makes it easy to deduce new knowledge from existing facts. For example, the inference engine is able to deduce to artifact from a blueprint. This gives us countless ways to infer even more information. Finally, the requirement is fulfilled.

5.2.3 Req-3: Action Selection

Summary Triggered by the rule engine, the orchestrator selects the appropriate actor. The actor is then responsible for executing the actual task.

Evaluation The heart of the deployment agent is orchestrator: every task and every event is either triggered by the rule engine or forwarded by the orchestrator. Hence, the select what to do next based on the knowledge have at that point of time.

5.2.4 Req-4: Reliable Deployment State

Summary The reconciler of the deployment agent ensures that the process list is either the previous or desired one. Thus, the deployment agent always keeps the system in a reliable deployment state.

Evaluation During the case study, we were able to confirm the always end in a valid deployment state.Thus, this requirement was successfully fulfilled by the reconciler.

5.2.5 Req-5: Flexibility

Summary Agent's sensors are extendable so a developer can easily implement custom sensors for their needs. By using the design abstraction, it is effortless to implement own process drivers.

Evaluation The flexibility was always the focus and it is possible to replace or extend any part of the deployment agent. For example, a developer could even add a new actor which modifies the process and blueprint. This kind of flexibility allows the deployment agent to run even on limited hardware.

6 Conclusion & Future Work

I don't know
if it's what you want,
but it's what you get. :-)

Larry Wall

Contents

6.1 Conclusion

This thesis presented the implementation of deployment agents. We discussed the primary goal of such a deployment agent. The primary goal is the reliable deployment of artifacts in a volatile environment. This was tackled by introducing the knowledge base. Although we stored facts with in the knowledge base, we quickly conclude that just storing them is not enough. Thus, we stated the inference engine deduced new knowledge from existing facts. Moreover, the ability to take action based on the knowledge gave us the rule engine and the orchestrator. Flexibility was a central goal because of the heterogeneous nature of IoT devices.
Lead by the requirement to be able to run devices with hardware limitation; we designed a flexible architecture. In addition, the case study shows us that we a more scalable than the current approach because we do not need to contact every device to do a deployment. With the deployment agent, it is the other way around.

All in all, we state that the deployment agent minimizes the risk of broken deployments because the agent puts the decision problem at the least possible time.

6.2 Future Work

There are countless possibilities for future work because of the architecture of the deployment and its focus on extensibility:

Blueprint As known from previous chapters, the blueprint sensor polls a GRPC endpoint in order to get the latest deployments. This GRPC endpoint is not common in

deployment pipeline. Therefore it makes sense to integrate another technology for the blueprint sensor such as plain http.

Driver Our driver approach allows us to integrate any kind of process. This enables great possibilities for more lightweight process manager. The following lists contains a few interesting drivers:

Containerd As mention in chapter 2, docker is based on containerd so it would make sense to implement that driver because the deployment agent does not need the overhead of docker.

Systemd Another possibility is to manage systemd task with the deployment agent.

Native Execution The most interesting driver implementation is the native execution driver which removes the entire overhead of the chosen container approach.

Rule Engine So far, the deployment agent has a simple rule engine. Nevertheless, there are many different variants of reasoning systems which can be evaluated by means of this deployment agent.

In conclusion, this list above shows exciting ideas to improve our deployment agent and integrate it into conventional continuous deployment systems.

Bibliography

[AF+15] A. Al-Fuqaha et al. “Internet of things: A survey on enabling technologies, protocols, and applications”. In: *IEEE Communications Surveys & Tutorials* 17.4 (2015), pp. 2347–2376 (cit. on p. 1).

[AIM10] L. Atzori, A. Iera, and G. Morabito. “The internet of things: A survey”. In: *Computer networks* 54.15 (2010), pp. 2787–2805 (cit. on pp. 3–5).

[Bib] K. J. Biba. *jail, section 9.* Accessed on 13.03.2018. URL: `https://docs.freebsd.org/44doc/papers/jail/jail-9.html` (cit. on p. 6).

[Dgr] (Cit. on p. 26).

[Doc] *Docker.* Accessed on 13.03.2018. URL: `https://www.docker.com/` (cit. on p. 26).

[Est17] P. Estes. *Multi-arch All The Things.* 2017. URL: `https://blog.docker.com/2017/11/multi-arch-all-the-things/` (cit. on p. 7).

[Eva11] D. Evans. “The internet of things: How the next evolution of the internet is changing everything”. In: *CISCO white paper* 1.2011 (2011), pp. 1–11 (cit. on pp. 1, 5).

[Fel+15] W. Felter et al. “An updated performance comparison of virtual machines and linux containers”. In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On.* IEEE. 2015, pp. 171–172 (cit. on pp. 7, 8).

[Gol] *The Go Programming Language.* Accessed on 13.03.2018. URL: `https://golang.org/` (cit. on p. 26).

[GYF12] G. Gruver, M. Young, and P. Fulghum. *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware.* Pearson Education, 2012 (cit. on p. 1).

[Har+16] A. Haroon et al. “Constraints in the IoT: the world in 2020 and beyond”. In: *Constraints* 7.11 (2016) (cit. on p. 11).

[HF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* 1st. Addison-Wesley Professional, 2010. ISBN: 0321601912, 9780321601919 (cit. on pp. 5, 6).

[Hyk17] S. Hykes. *Docker to donate containerd to the Cloud Native Computing Foundation.* 2017. URL: `https://blog.docker.com/2017/03/docker-donates-containerd-to-cncf/` (cit. on p. 8).

[Inc18] D. Inc. *Docker security*. Accessed on 13.03.2018. 2018. URL: https://docs.docker.com/engine/security/security/ (cit. on p. 7).

[INF4] D INFSO. "Networked Enterprise & RFID INFSO G. 2 Micro & Nanosystems". In: *Co-operation with the Working Group RFID of the ETP EPOSS, Internet of Things in* 2020 (4) (cit. on p. 3).

[MJL08] P. Menage, P. Jackson, and C. Lameter. "Cgroups". In: *Available on-line at: http://www. mjmwired. net/kernel/Documentation/cgroups. txt* (2008) (cit. on p. 7).

[Rod+17] P. Rodríguez et al. "Continuous deployment of software intensive products and services: A systematic mapping study". In: *Journal of Systems and Software* 123 (2017), pp. 263–291 (cit. on p. 1).

[Ros12] C. Rossi. *Ship early and ship twice as often*. Accessed on 13.03.2018. 2012. URL: https://de-de.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920/ (cit. on p. 1).

[Uni05] I. T. Union. "The Internet of Things—Executive Summary". In: *ITU Internet Reports* (2005) (cit. on p. 4).

[Wu+13] P. Wu et al. "Transition from IPv4 to IPv6: A state-of-the-art survey". In: *IEEE Communications Surveys & Tutorials* 15.3 (2013), pp. 1407–1424 (cit. on p. 5).

Glossary

CD Continuous Deployment.

cgroups .

CI Continuous Intergration.

CI/CD Continuous Integration and Continuous Deployment.

hypervisor A hypervisor creates and runs virtual machines..

IoT the Internet of Things.

IPC Inter-process communication.

IPv4 Fourth version of the Internet Protocol(IP) and it uses 32-bit addresses..

IPv6 Sixth version of the Internet Protocol(IP) and it uses 128-bit addresses..

kernel namespace Namespaces is a feature of Linux Kernel..

Object Naming Service A Object Naming Service translates RFID tags into IP addresses..

PID process ID.

RFID Radio-frequency identification.

TCP The Transmission Control Protocol (TCP) reliable, ordered delivery of bytes between two hosts communicating by an IP network..