Software and Firmware Updates for Internet of Things (WIP)

SIMON CARLSON

Master in Information Technology
Date: February 19, 2019
Supervisor: Shahid Raza, Farhad Abtahi
Examiner: Elena Dubrova
Swedish title: Mjukvaru- och Firmwareuppdateringar för Internet of Things
School of Computer Science and Communication

Abstract

E

Keywords: IoT, Industrial IoT, Security, Contiki, Embedded systems, Software updates

Sammanfattning

Svensk sammanfattning här.

Contents

List of Tables

List of Figures

Acronyms

ACE Authentication and Authorization for Constrained Environments

CA Certificate Authority

CoAP Constrained Application Protocol

DTLS Datagram Transport Layer Security

EST Enrollment over Secure Transport

HTTP Hypertext Transfer Protocol

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

PKI Public Key Infrastructure

SUIT Software Updates for Internet of Things

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

URI Uniform Resource Identifier

Chapter 1

Introduction

Internet of Things (IoT) is the notion of connecting physical objects to the world-spanning Internet in order to facilitate services and communication both machine-to-human and machine-to-machine. By having everything connected from everyday appliances to vehicles to critical parts of infrastructure, computing will be ubiquitous and the Internet of Things realized. The benefits from the IoT can range from quality of life services, such as controlling lights and thermostats from afar, to data gathering through wireless sensor networks, to enabling life critical operations such as monitoring a pacemaker or traffic system. IoT is a broad definition and fits many different devices in many different environments, what they all have in common is that they are physical and connected devices.

The Internet of Things is expected to grow massively in the following years as devices get cheaper and more capable. Ericsson expects the amount of IoT connections to reach 22.3 billion devices in 2024, of which the majority is short-range IoT devices [1]. However, cellular IoT connections are expected to have the highest compound annual growth rate as 4G and new 5G networks enable even more use cases for IoT devices.

In recent years the general public has become increasingly aware of digital attacks and intrusions affecting their day to day lives. In 2016 the DNS provider Dyn was attacked by a botnet consisting of heterogeneous IoT devices infected by the Mirai malware. Devices such as printers and baby monitors were leveraged to launch an attack on Dyn's services which affected sites like Airbnb, Amazon, and CNN [2]. Cardiac devices implanted in patients were also discovered to be unsafe, with hackers being able to deplete the batteries of pacemakers prematurely [3]. These cases and many others make it clear that security in IoT is a big deal, and as IoT is expected to grow rapidly insecure devices can cause even more problems in the future.

Security in IoT is also closely related to its business potential. According to Bain & Company, the largest barrier for Internet of Things adaptation is security concerns, and customers would buy an average of 70% more IoT devices if they were secured [4]. Despite security being lacking today in IoT, the field is expected to grow rapidly and an increase in unsecured devices could spell disaster. Securing IoT equipment such as printers, baby monitors, and pacemakers is imperative to prevent future attacks. But what about the devices currently employed without security, or devices in which security vulnerabilities are discovered after deployment? They need to be updated and patched in order to fix these vulnerabilities, but sending a technician to each and every of the predicted 22.3 billion devices is not feasible. They need secure and remote updates which is a non-trivial task.

1.1 Problem Statement

There is a need for secure software and firmware updates for IoT devices as vulnerabilities must be patched. For many IoT devices this mean patches must be applied over long distances and possibly unreliable communication channels as sending a technician to each and every device is unfeasible. There are non-open, proprietary solutions developed for specific devices but no open and interoperable standard. The Internet Engineering Task Force (IETF) Software Updates for Internet of Things (SUIT) working group aims to define an architecture for such a mechanism without defining new transport, discovery, or security mechanisms [5]. By expanding upon the work of the SUIT group, a standardized update mechanism suitable for battery powered, constrained, and remote IoT devices can be developed.

1.1.1 Problem

The thesis project will examine the architecture and information model proposed by SUIT in order to create and evaluate an updating mechanism for battery powered, constrained, and remote IoT devices with a life span exceeding five years. The update mechanism can be considered in three parts: communication, upgrading, and device management. Communication will happen over unreliable channels and the payloads must arrive intact and untampered with. The upgrade mechanism itself must ensure integrity of the target image and safely perform the upgrade with a limited amount of memory. Device management concerns managing devices in a network and seeing which devices are in need of updates.

The thesis will examine the update mechanism from the viewpoint of IoT devices running the Contiki-NG operating system on 32-bit ARM Cortex M3 processors. A suitable public key infrastructure developed by RISE will also be an underlying assumption for the work. The thesis aims to investigate the problem "How can the SUIT architecture be used to provide secure software and firmware update for the Internet of Things?".

1.1.2 Purpose

The purpose of the thesis project is to provide an open and interoperable update mechanism that complies with the standards suggested by the IETF SUIT working group. This will aid other projects trying to secure their IoT devices following accepted standards.

1.1.3 Goal

The goals of the thesis are to study current solutions and technologies and propose a technology agnostic update architecture based on the SUIT architecture, as well as proposing lightweight end-to-end protocols that instantiates the architecture, allowing updates of IoT devices. The degree project shall deliver specifications and prototypes of the protocols in an IoT testbed, as well as a thesis report.

1.2 Methodologies

The thesis will follow a mix of qualitative and quantitative methodologies. The SUIT group defines some goals or constraints a suitable updating mechanism should follow, but as their proposed architecture is agnostic of any particular technology these goals cannot be easily quantified. It is better to regard them as qualitative properties the mechanism should have. In addition there are some relevant measurements, such as reliability of the communication and memory and power requirements of the updates, that can be used in an evaluation. The quantitative part of the evaluation will follow an objective, experimental approach, while the qualitative part will follow a interpretative approach.

1.3 Risks, Consequences, and Ethics

As IoT devices become more commonplace their security becomes more important. Incorrect or faulty firmware or software can lead to incorrect sensor readings, a common application of IoT devices, which in turn can lead to incorrect conclusions. This could have a large effect on businesses such as agriculture and healthcare. Insecure communication channels can expose confidential or personal data, something governments and international unions are taking more seriously.

As these devices can be connected on networks with other computers such as laptops, a compromised IoT device can cause an attack to proliferate throughout an entire network. This can affect other IoT devices as well as traditional computers, putting every connected device at risk. Furthermore hacked devices can leak data to attackers and cause service disruptions. All of these risks and more have to be accounted for as IoT is expected to boom.

1.4 Scope

The thesis will primarily focus on the use case of applying an entire update to one single device consisting of only one microcontroller. The use case of applying differential updates is considered important and the architecture will enable these kinds of updates, but the prototype will only support it with respect to time.

1.5 Outline

Will be corrected when the outline is set.

Chapter 2

Background

This chapter presents the background needed to understand the thesis. Section 2.1 introduces the network protocols used and motivates their use over other protocols in an IoT context. The following section, Section 2.2 presents and explains the SUIT architecture and information model and their respective requirements as formulated by the IETF. Finally, Section 2.3 presents the Contiki-NG operating system which the updating mechanism will be developed for as well as the target hardware.

2.1 IoT Network Stack

Network protocols in IoT networks operate under different circumstances compared to traditional computer networks. Whereas traditional networks enjoy high reliability, high throughput, and high computational performance IoT networks are defined by their low power requirements, low reliability, and low computational performance on edge devices. This posts some constraints on the protocols used in IoT as they must properly handle these characteristics.

One of the most widely used network protocol stacks today in traditional networks is based on the Internet Protocol (IP). The stack uses Transmission Control Protocol (TCP) as a transportation protocol, usually with Transport Layer Security (TLS) for security, and a common application layer protocol is Hypertext Transfer Protocol (HTTP). TCP is however poorly suited for IoT networks as it is a connection based, stateful protocol which tries to ensure the guaranteed delivery of packets in the correct order. There is also advanced congestion control

IoT stack			Traditional stack	
CoAP(s)		$\Longleftrightarrow$	HTTP(S)	
UDP	DTLS	$\Longleftrightarrow$	TCP	TLS
IPv6		$\Longleftrightarrow$	IPv4 or IPv6	
IEEE 802.15.4	6LoWPAN	$\Longleftrightarrow$	IEEE 802.15.4	
IEEE 802.15.4		$\Longleftrightarrow$	IEEE 802.15.4	

Figure 2.1: Comparison of network stacks between IoT networks and traditional networks.

mechanisms in TCP which are hard to apply on unreliable low-bandwidth networks. IoT networks often utilize User Datagram Protocol (UDP) as a transport protocol instead. UDP is also an IP-based protocol but is connectionless and less reliable than TCP, performing on a best-effort level instead. Despite being less reliable, UDP often performs better in IoT contexts.

As TLS is based on the same assumptions as TCP it is unsuitable for UDP networks. UDP networks still need confidentiality and integrity through some means and thus use Datagram Transport Layer Security (DTLS), which is a version of TLS enhanced for use in datagram oriented protocols. HTTP can be used over UDP for the application layer, but as HTTP is encoded in human readable plaintext it is unnecessarily verbose and not optimal for constrained networks. Instead, Constrained Application Protocol (CoAP) is a common protocol for the application layer in IoT networks. Figure 2.1 shows the equivalent protocols for IoT network stacks versus traditional network stacks.

In this chapter, Section 2.1.1 explains UDP and why it is the preferred transport protocol in IoT networks. Section 2.1.2 briefly explains TLS, why it is unsuitable for IoT networks, the differences between TLS and DTLS and why DTLS is used instead. Section 2.1.3 describes the CoAP protocol. Lastly, Sections 2.1.4 and 2.1.5 briefly introduces the EST-coaps protocol and ACE framework, enabling asymmetric cryptography and authorization in an IoT context.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Source port	Destination port
Length	Checksum
Data octets . . .	

Figure 2.2: The UDP header format.

2.1.1 UDP

UDP is a stateless and asynchronous transfer protocol for IP [6]. It does not provide any reliability mechanisms but is instead a best-effort protocol. It also does not guarantee delivery of messages. For general purposes in unconstrained environments TCP is usually the favored transport protocol as it is robust and reliable, but in environments where resources are scarce and networks unreliable, a stateful protocol like TCP could face issues. Since TCP wants to ensure packet delivery, it will retransmit packages generating a lot of traffic and processing required for a receiver. Also, if the connection is too unstable TCP will not work at all since it can no longer guarantee the packets arrival. The best-effort approach of UDP is favorable in these situations, in addition to UDP being a lightweight protocol requiring a smaller memory footprint to implement.

Figure 2.2 shows the UDP header format. The source port is optional, the length denotes the length of the datagram (including the header), and the checksum is calculated on a pseudo-header constructed from both the IP header, UDP header, and data. UDP headers are minimalist which shows the point of it being a lightweight protocol with not many features available.

2.1.2 DTLS

DTLS is a protocol which adds confidentiality and integrity to datagram protocols like UDP [7]. The protocol is designed to prevent eavesdropping, tampering, or message forgery. DTLS is based on TLS, a similar protocol for stateful transport protocols such as TCP, which would not work well on unreliable networks as previously discussed. The main issues with using TLS over unreliable networks is that TLS decryption is dependant on previous packets, meaning the decryption of a packet would fail if the previous packet was not received. In addition, the TLS handshake procedure assumes all handshake messages are delivered reliably which is rarely the case in IoT networks using UDP.

DTLS solves this by banning stream ciphers, effectively making decryption an independent operation between packets, as well as adding explicit sequence numbers. Furthermore, DTLS supports packet retransmission, reordering, as well as fragmenting DTLS handshake messages into several DTLS records. These mechanisms makes the handshake process feasible over unreliable networks.

By splitting messages into different DTLS records, fragmentation at the IP level can be avoided since a DTLS record is guaranteed to fit an IP datagram. IP fragmentation is problematic in low-performing networks since if a single fragment of an IP packet is dropped all fragments of that packet must be retransmitted, and thus fragmenting at the IP level should be avoided. Since DTLS is designed to correctly handle reordering and retransmission in lossy networks, splitting messages into several DTLS records is no problem, and if one record is lost only that record needs to be retransmitted in a single IP packet.

In order to communicate via TLS and DTLS, a handshake has to be carried out. The handshake establishes parameters such as protocol version, cryptographic algorithms, and shared secrets. The TLS handshake involves hello messages for establishing algorithms, exchanging random values, and checking for earlier sessions. Then cryptographic parameters are shared in order to agree on a shared premaster secret. The parties authenticate each other via public key encryption, generate a shared master secret based on the premaster secret, and finally verifies that their peer has the correct security parameters.

The DTLS handshake adds to this a stateless cookie exchange to complicate DoS attacks, some modifications to the handshake header to make communication over UDP possible, and retransmission timers since the communication is unreliable. Otherwise the DTLS handshake is as the TLS handshake.

2.1.3 CoAP

CoAP is an application layer protocol designed to be used by constrained devices over networks with low throughput and possibly high unreliability for machine-to-machine communication [8]. While designed for constrained networks, a design feature of CoAP is how it is easily interfaced with HTTP so that communication over traditional networks can be proxied. CoAP uses a request/response model similar to HTTP with method codes and request methods that are easily mapped to those of HTTP. Furthermore CoAP is a RESTful protocol utilizing concepts such as endpoints, resources, and Uniform Resource Identifier (URI). These features makes it easier to design IoT applications that seamlessly interact with traditional web services. Additionally CoAP offers features such as multicast support, asynchronous messages, low header overhead, and UDP and DTLS bindings which are all suitable for constrained environments.

As CoAP is usually implemented on top of UDP, communication is stateless and asynchronous. For this reason CoAP defines four message types: Confirmable, Non-confirmable, Acknowledgment, and Reset. Confirmable messages must be answered with a corresponding Acknowledgment, this provides one form of reliability over an otherwise unreliable channel. Non-confirmable messages do not require an Acknowledgment and thus act asynchronously. Reset messages are used when a recipient is unable to process a Non-confirmable message.

Since CoAP is based on unreliable means of transport, there are some lightweight reliability and congestion control mechanisms in CoAP. Message IDs allows for detection of duplicate messages and tokens allow asynchronous requests and responses be paired correctly. There is also a retransmission mechanism with an exponential back-off timer for Confirmable messages so that lost Acknowledgments does not cause a flood of retransmissions. Additionally, CoAP features piggybacked responses, meaning a response can be sent in the Acknowledgment of a Confirmable or Non-Confirmable request if the response fits and is available right away. This also reduces the amount of messages sent by the protocol.

The length of the payload is dependant on the carrying protocol and is calculated depending on the size of the CoAP header, token, and options as well as maximum DTLS record size. Section 4.6 of the CoAP standard says "If the Path MTU [Maximum Transmission Unit] is not known for a destination, an IP MTU of 1280 bytes SHOULD be assumed; if nothing is known about the size of the headers, good upper bounds are 1152 bytes for the message size and 1024 bytes for the payload size" [8].

Since firmware images can be relatively large their size needs to be handled during transportation, which can be done via block-wise transfers [9]. A Block option allows stateless transfer of a large file separated in different blocks. Each block can be individually retransmitted and by using monotonically increasing block numbers, the blocks can be reassembled. The size of blocks can also be negotiated between server and client meaning they can always find a suitable block size, making the mechanism quite flexible. Another option of interest is the observe option, which allows a client to be notified by the server when a particular resource changes. This option can be used in a pull or hybrid update architecture, meaning the device does not have to continuously poll the server for a new update.

2.1.4 EST-coaps

EST-coaps is a protocol for certificate enrollment in constrained environments using CoAP. [10]. By allowing IoT devices to enroll for certificates, asymmetric encryption can be used even in a constrained environment. EST-coaps is heavily based on Enrollment over Secure Transport (EST) which was developed for traditional, less constrained networks and is thus incompatible with the SUIT standard, which is specified to work on very small devices [11]. EST-coaps retains much of the functionality and structure of EST but modifies it slightly to work over CoAP, DTLS, and UDP instead of HTTP, TLS, and TCP, for instance by making use of CoAPs block requests and responses to remedy the relatively large sizes of certificates.

2.1.5 ACE

Authentication and Authorization for Constrained Environments (ACE) is a framework for authorization and authentication for IoT contexts, based on the OAuth 2.0 framework [12]. ACE allows for clients in a network to access protected resources through authorization tokens. An authorization server is used to distribute tokens to clients, which they then pass on to resource servers. If a token is valid and the resource requested matches the level of authority associated with the token, access to the resource is granted. The framework is flexible regarding topologies, and clients can independently verify the validity of tokens or ask the authorization server for more information through an introspection request.

The ACE framework can be used to authorize clients in an IoT context. Authorization is important and distinct from identification, which can be achieve through for instance a PKI. As the architecture proposed in this thesis aims to be standardized it should prepare for different parties assuming different roles in the context of updating devices. Using authorization tokens is a scalable and configurable way of achieving that.

2.2 SUIT

The IETF Software Updates for Internet of Things (SUIT) working group aims to define a firmware update solution for IoT devices that is interoperable and non-proprietary [5]. The working group does not however try to define new transport, discovery, or security mechanisms making their proposal agnostic of any particular technology. The solution shall be usable on Class 1 devices, defined by 10 KiB RAM and 100 KiB code size [13]. The solution may be applied to more powerful devices, such as the one used in this thesis described in Section 2.3.1.

Section 2.2.1 presents the SUIT architecture and Section 2.2.2 presents the SUIT information model.

2.2.1 Architecture

There is an Internet Draft by the SUIT group focusing on the architecture of an IoT update mechanism [14]. This draft describes the goals and requirements of such an architecture, although makes no mention of any particular technology. The overarching goals of the update process is to thwart any attempts to flash unauthorized, possibly malicious firmware images as well as protecting the firmware image's confidentiality and integrity. These goals reduces the possibility of an attacker either getting control over a device or reverse engineering a malicious but valid firmware image as an attempt to mount an attack.

In order to accept an image and update itself, a device must make several decisions about the validity and suitability of the image. The information needed comes in form of a manifest. The next section will describe the requirements posted upon this manifest in more detail. The manifest helps the device make important decisions such as if it trusts the author of the new image, if the image is intact, if the image is applicable, where the image should be stored and so on. This in turns means the device also has to trust the manifest itself, and that both manifest and update image must be distributed in a safe and trusted architecture. The draft [14] presents ten qualitative requirements this architecture should have:

- Agnostic to how firmware images are distributed:
  The mechanism should not assume a particular technology is used to distributed manifests and images, but instead be able to be carried over different mediums. This means decisions about formats and distribution methods must not rely on features of a particular technology.

- Friendly to broadcast delivery:
  The mechanism should be broadcast friendly, meaning the mechanism can not be reliant on security on the transport layer or below. Also, devices receiving broadcast updates not meant for them should not incorrectly apply the update.

- Use state-of-the-art security mechanisms:
  The SUIT standard assumes a Public Key Infrastructure (PKI) is in place. The PKI will allow for signing of the update manifest and firmware image.

- Rollback attacks must be prevented:
  The manifest will contain metadata such as monotonically increasing sequence numbers and best-before timestamps to avoid rollback attacks.

- High reliability:
  The act of upgrading an image should not cause the device to break. This is an implementation requirement.

- Operate with a small bootloader:
  The bootloader should be minimal. This is also an implementation requirement.

- Small parser:
  It must be easy to parse the fields of the update manifest as large parser can get quite complex. Validation of the manifest will happen on the constrained devices which further motivates a small parser and thus less complex manifests.

- Minimal impact on existing firmware formats:
  The update mechanism itself must not make assumptions of the current format of firmware images, but be able to support different types of firmware image formats.

- Robust permissions:
  Updates must be authorized before they are applied, and different configurations might have different requirements for authorization. The architecture should enable a flexible and robust permission model.

- Operating modes:
  The draft presents three broad modes of updates: client-initiated updates, server-initiated updates, and hybrid updates, where hybrids are mechanisms that require interaction between the device and firmware provider before updating. The thesis will look into all three of these broad classes. Some classes may be preferred over others based on the technologies chosen in the thesis.

The distribution of manifest and firmware image is also discussed, with a couple of options being possible. The two approaches are shown in Figure 2.3 and Figure 2.4. The first figure shows the manifest and image distributed together to a firmware server. The device then receives the manifest either via pulling or pushing and can subsequently download the image from the same server. Alternatively, as shown in the second figure, the manifest itself can be directly sent to the device without a need of a firmware server, while the firmware image is put on the firmware server. After the device has received the lone manifest through some method, the firmware can be downloaded from the firmware server. The SUIT architecture does not enforce a specific method to be used when delivering the manifest and firmware, but states that an update mechanism must support both types.

2.2.2 Information Model

The Internet Draft for the SUIT information model presents the information needed in the manifest to secure a firmware update mechanism [15]. A manifest is needed for a device to make a decision about whether or not to update itself, and if the image related to the manifest is valid and its integrity ensured. The draft also presents threats,

Figure 2.3: Distributing both manifest and image through a firmware server.

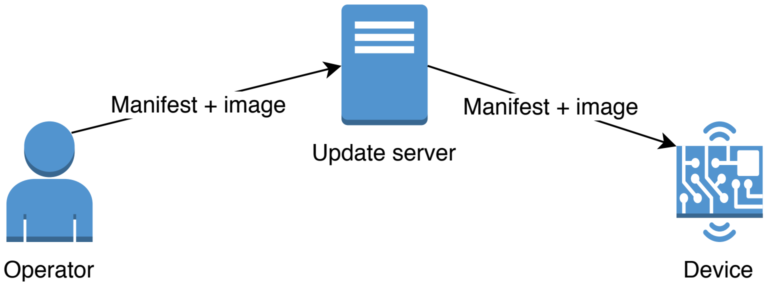
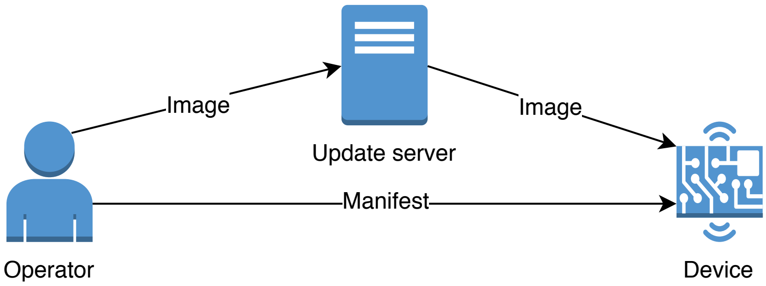


Figure 2.4: Distributing the manifest directly to the device and image through a firmware server.



classifies them according to the STRIDE model, and presents security requirements that map to the threats [16]. Finally it presents use cases and maps usability requirements to the use cases in order to motivate the presence of each manifest element. Note that the information model does not discuss threats outside of transporting the updates, such as physical attacks.

The proposed mandatory and recommended manifest elements and their brief motivations can be seen in Table 2.1. For the optional elements and more detailed motivations, use cases, and requirements refer to [15].

Table 2.1: The proposed mandatory and recommended manifest elements of the SUIT information model. Adapted from [15].

Table 2.1: The proposed mandatory and recommended manifest elements of the SUIT information model. Adapted from [15].

Manifest Element	Status	Motivation/Notes
Version identifier	Mandatory	Describes the iteration of the manifest format
Monotonic Sequence Number	Mandatory	Prevents rollbacks to older images
Payload Format	Mandatory	Describes the format of the payload
Storage Location	Mandatory	Tells the device which component is being updated, can be used to establish physical location of update
Payload Digest	Mandatory	The digest of the payload to ensure authenticity. Must be possible to specify more than one payload digest.
Size	Mandatory	The size of the payload in bytes
Signature	Mandatory	The manifest is to be wrapped in an authentication container (not a manifest element itself)
Dependencies	Mandatory	A list of digest/URI pairs linking manifests that are needed to form a complete update

Table 2.1: The proposed mandatory and recommended manifest elements of the SUIT information model. Adapted from [15].

Manifest Element	Status	Motivation/Notes
Precursor Image Digest Condition	Mandatory (for differential updates)	If a precursor image is required, this digest condition is needed
Content Key Distribution Method	Mandatory (for encrypted payloads)	Tells how keys for encryption/decryption are distributed
Vendor ID Condition	Recommended	Helps distinguish products from different vendors
Class ID Condition	Recommended	Helps distinguish incompatible devices in a vendors infrastructure

As the SUIT architecture tries to be a standardized solution it must account for different use cases and different combinations of use cases. As a result many of the mandatory and recommended elements are there to enable certain use cases that might not always be relevant. This means certain information must be prepared for in advance even if it is not going to be used in all cases. There is a trade off with flexibility and size, and as the devices of interest are lightweight it is of interest to reduce the size of the manifest as much as possible without limiting use cases. With these two considerations in mind, a manifest for the architecture proposed in this thesis must be designed to facilitate as many different use cases as possible while keeping sizes to a minimum.

To summarize, the SUIT information model proposes to use a signed manifest that is distributed to each device in need of an update through some method. The device then parses the manifest in order to determine if the update is trusted, suitable, and up to date, with many other optional elements such as if special processing steps or new URIs to fetch the images are needed. The model does not make assumptions about technology which is one of the reasons there are optional elements, not all of them are applicable to all solutions. Nevertheless, the architecture and information model together provides a solid base on which to design a secure update mechanism for IoT.

2.3 Contiki-NG

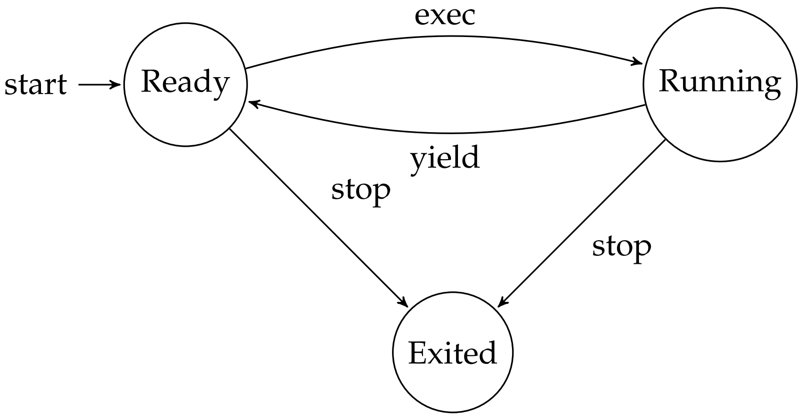
Contiki-NG is an open-source operating system for resource constrained IoT devices based on the Contiki operating system [17, 18]. Contiki-NG features an IPv6 network stack designed for unreliable, low-power IoT networks. There are many protocols implemented in the stack, this thesis will look at UDP and CoAP secured by DTLS. Beneath IPv6 Contiki-NG supports IEEE 802.15.4 with Time Slotted Channel Hopping [19].

The CoAP implementation in Contiki-NG is based on Erbium by Mattias Kovatsch and supports both unsecured (CoAP) and secured (CoAPs) communication [20]. CoAPs uses a DTLS implementation called TinyDTLS which handles encryption and decryption of messages [21].

Contiki-NG has a process abstraction which is built on lightweight protothreads [22]. Protothreads can be seen as a combination of threads and event-driven programming. Threads provide a way of running sequential programs concurrently, enabling the use of flow-control mechanisms. This however requires a thread-local stack which is too resource heavy on small IoT devices. Event-driven programming on the other hand does not require a thread-local stack but programs are limited to computation as callbacks on event triggers, meaning sequential programs are more difficult to program. Protothreads combine these paradigms in a lightweight way, keeping the yield semantics of threads and stacklessness of event-driven programming. All protothreads in a system are run on the same stack, meaning each protothread has a very low memory overhead, and by providing a conditional blocking wait statement protothreads can execute cooperatively.

Contiki-NGs execution model is event based, meaning processes often yield execution until they are informed a certain event has taken place, upon which they can act. Figure 2.5 shows the states of the processes and their transitions. User-space processes are run in a cooperative manner while kernel-space processes can preempt user-space processes. Examples of events are timers expiring, a process being polled, or a network packet arriving.

Figure 2.5: The state machine of Contiki threads. Adapted from [23].



Contiki-NG provides two memory allocators in addition to using static memory. They are called `MEMB` and `HeapMem` and are semi-dynamic and dynamic, respectively. `MEMB` is a semi-dynamic allcator that allocates pools of static memory as arrays of constant sized objects. After a memory pool has been declared, it is initialized after which objects can be allocated memory from the pool. All objects allocated through the same pool have the same size. `HeapMem` solves the issue of dynamically allocating objects of varying sizes during runtime in Contiki-NG. It can be used on a variety of hardware platforms, something a standard C `malloc` implementation could struggle with. With `HeapMem` memory can be reallocated and deallocated as if using a normal `malloc`.

2.3.1 Firefly

Zolertia Firefly is a supported hardware platform in Contiki-NG and the targeted board for this thesis. It is a breakout board designed for IoT application development sporting an ARM Cortex-M3 with 512 KB flash and 32 KB RAM, making it more powerful than the Class 1 devices the SUIT standard specifies as a lower bound. This is not an issue but should be kept in mind as less capable devices are supposed to be able to use an update mechanism that is complying with SUIT. Furthermore the board supports IEEE 802.15.4 communication in the 2.4 GHz band and SHA2 and RSA hardware acceleration [24].

Chapter 3

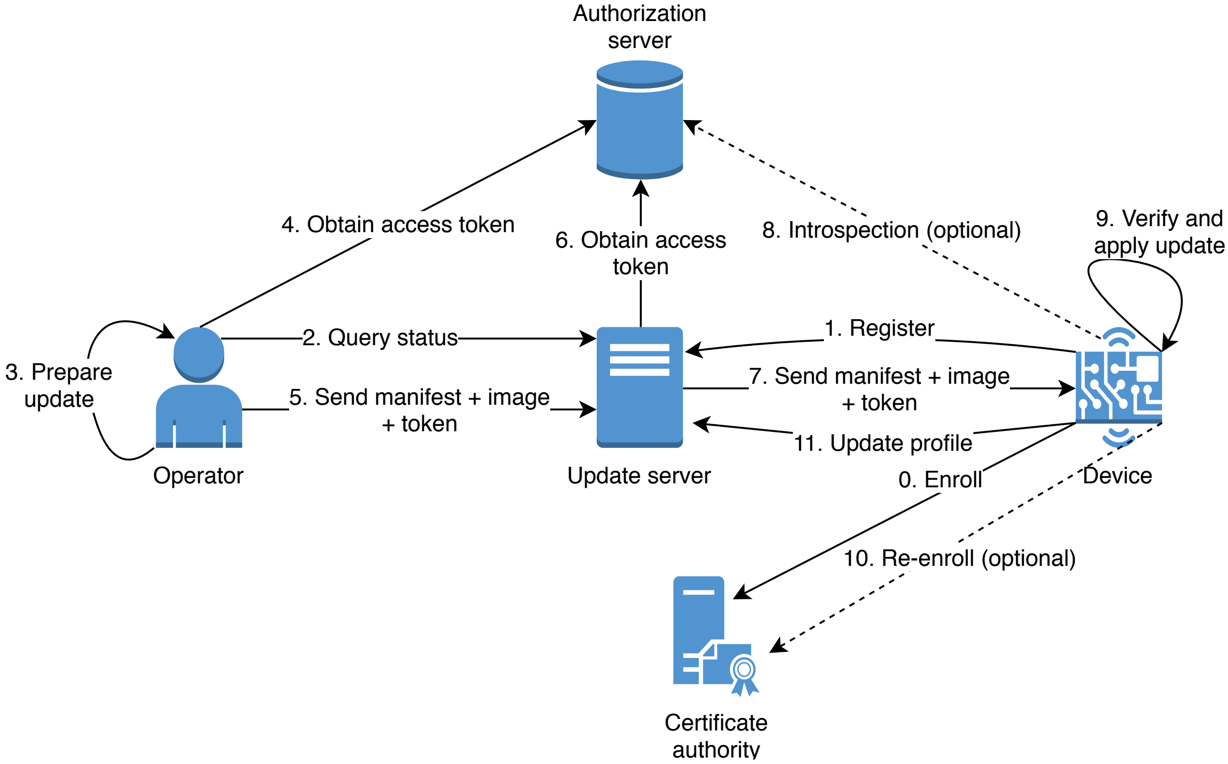
Software Updates for Internet of Things Architecture

This thesis proposes an update architecture based on SUIT. The architecture is technology agnostic and adds on top of SUIT tokens and certificates for identifying and authorizing actors, a life cycle perspective on device management, and defines what information is needed for a device to take part of an update procedure. Furthermore the thesis presents profiles for the architecture which implementers can use, these are introduced in Chapter 5. Five key areas that the architecture must define have been identified:

- Roles of devices, servers, and operators
- Key management of IoT update procedure
- Device profiles and communication
- Update authorization
- Update handling for local upgrades

By putting the five key areas together, Figure 3.1 shows an example update flow in the architecture. Note that where the SUIT standard would differentiate between the author of an update and an operators, this architecture will for simplicity regard them as the same actor. For the same reason, the architecture will not differentiate between device and network operators. An operator is an actor that tracks device statuses and prepares, signs, and sends updates to devices. Figure 3.1

Figure 3.1: Example workflow of an update procedure.



gives a brief introduction to the concepts needed to understand the architecture, these concepts will be explained in more detail throughout sections 3.1-3.4. In this particular example the update procedure is initiated by the operator by querying the server for device status and then pushing an update. The manifest and image are distributed together. The steps can be briefly explained as:

0. The device enrolls at the CA and receives a certificate.
1. The device registers at the server which now holds a profile for the device.
2. The operator queries the server for device status in order to prepare an update.
3. The operator prepares a manifest and image and signs them.
4. The operator requests an authorization token from the authorization server in order to gain access to applying the update.
5. The signed manifest and image are sent to the server with the authorization token.
6. The server requests an authorization token from the authorization server in order to gain access to applying the update.
7. The signed manifest and image are sent with the server's authorization token to the device.
8. The device requests introspection data from the authorization server to verify the authorization token. This step is optional.
9. The device decrypts and verifies the manifest and image and applies the update.
10. The device's certificate broke as part of the update and it uses a new pre-shared key included in the updated image to re-enroll. This step is optional.
11. The device updates its profile at the server by re-registering.

Section 3.1 defines what a device, server, and operator means in the context of the update architecture. Section 3.2 discusses what is needed for devices to enroll in a PKI and how keys are handled during updates. Section 3.3 describes how devices, servers, and operators can communicate and shows an example workflow of communicating an update. Section 3.4 describes the purpose of issuing authorization tokens, how devices can use them, and who is to be authorized. Section 3.5 discusses different means of handling the payload when applying an update. Finally Section 3.6 shows the architecture applied to different use cases.

3.1 Roles of Devices, Servers, and Operators

This section explains the notion of devices, servers, and operators in the architecture, their responsibilities, and their functionality. Section 3.1.2 covers the topic of servers, Section 3.1.3 covers operators, and Section 3.1.1 briefly covers devices.

3.1.1 What Is a Device and What Do They Do?

Devices are constrained, low-power IoT appliances connected to a constrained network. They are running the applications of the network and performs simple tasks such as measurement using sensors. The devices communicate wirelessly and must be secured from attackers while being able to be updated. They communicate with each other, servers, and operators.

Communication with servers and operators requires the servers and operators know how to reach the devices if they want to initiate contact. Devices must thus register profiles at servers, and must be enrolled in order to be trusted. In order to register a profile at a server, the server must offer an API that allows a device to POST its information (vendor and class ID as well as version) to the server.

Devices have to trust servers and operators wanting to send them updates. In order to do so, devices need predetermined whitelists of servers and operators. The identity of a server or operator can be checked using certificates and by then matching against the whitelists, communication can be allowed or prohibited. Servers and operators that are whitelisted still need to be authorized to perform an update, this is discussed further in Section 3.4.

To properly handle updates a device needs a local update handler. The update handler is responsible for decrypting manifests and images, parsing and verifying the manifest, and preparing the image for boot. How and where the device stores manifest and image is discussed in further detail in Section 3.5.

3.1.2 What Is a Server and What Do They Do?

Servers are responsible for transporting manifests and images to the devices, acting as image repositories, and keeping track of device profiles. After enrollment devices register at a server and the server creates a profile for that device. The profile contains the vendor and class IDs and firmware version of the device. This allows the server to know through which protocols the device can be contacted and which version it should be updated to. Operators, discussed in the next section, send signed manifests and images to servers, and can query them for device status.

Devices should contain a list of servers and by trusting the certificate authority they can validate server certificates. A server is thus any machine that is enrolled, has a valid certificate, and is included in the device's list of servers. The reasoning behind this definition is that a standard solution for updates should not assume the topology of an IoT network. The server may be a machine acting as a proxy between a traditional network containing the operator and a constrained network with IoT devices. The server could be a more capable IoT device located entirely within the constrained network and be contacted through a proxy. The server could be located entirely within the traditional network and use a proxy to communicate with devices of the constrained network.

A device can be aware of several servers, and different devices can be mapped to different servers. This can make device management easier as certain classes of devices can be handled by certain servers. By allowing a device to receive updates from several servers, the update mechanism architecture also displays a form of robustness. If one server is for instance located entirely within the constrained network and the connection between that server and the operator is severed, updates can still be distributed through other servers. If devices are pulling updates, they can query the servers in order of their list of servers. If updates are pushed, devices keep the connection with the server pushing the update.

Which machines are allowed to act like servers can be boiled down to a few important points no matter the topology of choice:

- The server is enrolled and has a valid certificate
- The server is included in a device's list of servers (meaning it is authorized to send updates to the device)
- An operator can reach the server and is authorized by the server to upload updates

3.1.3 What Is an Operator and What Do They Do?

Operators are people authorized to upload manifests and images to a server. They can also optionally upload manifests directly to a device depending on the architecture. Operators prepare manifests and images, signs and transports them with an authorization token to a server, which then forwards them to a device. The signing ensures end-to-end security for images and manifests between operators and devices. Authorization is further discussed in Section 3.4.

Like a list of servers introduced in Section 3.1.2, devices also need a list of operators. This is because if an operator wishes to send a manifest directly to a device, the device needs to be aware of the operator and permit traffic from that operator. Just like with mapping devices to servers, devices can receive manifests from several operators and different devices may interact with different operators. This creates opportunities to logically divide a network between different operators. An example use case is a constrained network supported by different vendors where the respective vendors should only be able to service their respective devices. It can also create a hierarchy, where certain operators may directly interact with devices but other operators must interact with devices through servers. Yet again the point is to prepare for as many different scenarios as possible and create a flexible architecture.

As with servers, the essence of being an operator can be captured in a few points:

- An operator is enrolled and has a valid certificate

- An operator is included in a device's list of operators (and therefore can send manifests to the device)
- An operator is trusted by a server to send manifests and images to the server

3.2 Key Management of IoT Update Procedure

The architecture will, in order to align with the goals of SUIT, be based on asymmetric cryptography. The availability of EST-coaps makes this feasible in IoT contexts but other enrollment protocols could also be used. This means a Certificate Authority (CA) is needed to act as a trusted third party distributing certificates. The certificates are linked to a public key which has a private key partner and are used to verify the correctness of a public key. Certificates are signed by the CA and in order to trust them, device have to trust the CA itself.

In order to enroll, a pre-shared key is proposed. This is the approach used in EST-coaps, but it is not chosen for that reason. If a pre-shared key is not used the CA cannot be sure the device asking to enroll really should be part of the trusted network. If an attacker obtains a valid certificate they could communicate with devices and servers alike and no one would be able to tell it is a malicious actor. CAs must know they are issuing certificates to the correct devices and pre-shared keys gives devices a means of identifying themselves. Pre-shared keys could be used for encrypting all traffic but as they are less scalable and harder to manage than certificates, they are just used for enrolling.

A device that is enrolling has to trust the CA issuing the certificate. If it cannot do so, how could it know it received a valid certificate? An attacker would love for a device to use the attacker's public key instead, and if an attacker poses as a CA it could issue a certificate with its own public key and then sign it. In order to trust the CA, a device needs to have the certificate of the CA it is enrolling with or some other CA further down the chain of trust. By verifying the signature on the newly enrolled certificate with the CAs own certificate, a device can be certain they are using the correct keys.

When applying updates, certificates might not be valid anymore.

This is implementation dependant and might not always be a problem, but the architecture should be prepared for these situations. In the case an update breaks a certificate, the certificate cannot be used for communication and thus the device cannot use it to re-enroll either. In this case, a new pre-shared key should be part of the update so that the device can enroll as if it was factory new. The process of issuing pre-shared keys might be difficult to automate as the CA must be aware of which keys to accept, but it is needed to ensure the security of future device communication.

3.3 Device Profiles and Communication

In heterogeneous networks of IoT devices, each device might sport a specific protocol stack. In order to enable different devices to be updated, servers must be aware of how to reach these devices. This problem introduces the notion of device profiles containing information about device capabilities and software versions.

When devices have enrolled and obtained a valid certificate, they must contact their respective servers so the servers can create profiles of the devices. The profiles will tell through which protocols to reach a device and what software versions the device is using. In order to achieve this, devices must first know which servers to contact. This can be solved through shipping devices with a list of servers. This list can later on be updated like any other software. Furthermore, the devices use vendor and class IDs as described in the SUIT information model. The information model uses these pieces of information to verify an update is intended for a specific device by matching the IDs, but they can also be sent to a server to tell it what kind of device is contacting it. The server can infer a profile based on the IDs it is sent, or simply use the protocols the device chose to contact it. The devices also need to be aware of how to register, for instance by POSTing to a specific server endpoint.

When updating there are possibilities regarding how the updating process is initiated. An operator can query the server for the status of one or several devices, prepare an update for them, and have the update pushed through the server. Optionally the operator could send a manifest to the device explaining when the update is to be applied and put the image on the server. Later when the device should update, the device pulls the image from the server, validates it using the already received manifest, and updates. Both the pull and push approaches assume the devices are already enrolled and registered at the server.

After updating, the device's capabilities might have changed, for instance by having a new protocol implemented in software. The version has also changed due to applying an update. After an update, a device should notify all its servers so they can update the device profile (or simply discard the old one and generate a new, as if the device registered for the first time). This ensures the servers view of the devices is up to date and that communication will always happen through the intended and supported protocols. The functionality of registering is the same as when a device is new and thus not costly to implement.

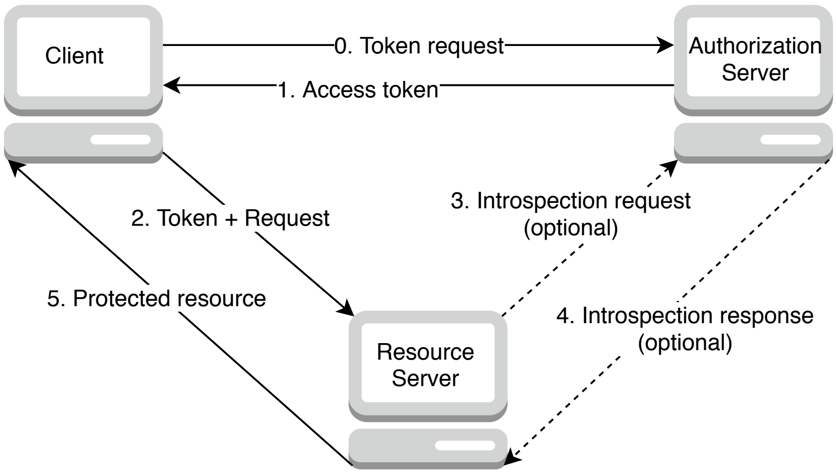
There are alternatives to using device profiles. One alternative is to instead keep a list of known protocols implemented by devices in the network, and when pushing updates to a device trying each protocol in sequence. This has the benefit of not needing to keep and continuously update profiles, but also has some issues. One issue is that you still need to keep some state of the devices on the server regarding firmware version. If the server does not know what the status of device versions are, it cannot help a human operator decide about deploying updates.

Another drawback is that devices might implement common protocols but have different preferences. If two devices implement some common protocols but one of them supports hardware operations for encrypting one of the protocols, it will prefer using that protocol with the server, whereas the other device might not. The server will however, without information about device preference, try the same sequence of protocols with both devices.

Furthermore, as communication can be unreliable over these networks, the server cannot know for sure if a device does not respond due to not understanding the protocol and therefore dropping the packets, or if the response just got lost in transmission. It is more robust to keep track of which protocols devices support and conform to the preferences of the constrained devices.

Lastly, instead of using profiles all communication could be initiated from the device side, meaning updates are only done through a pulling mechanism. This would not enable the use case of pushing updates which could be critical if a vulnerability must be patched right

Figure 3.2: The protocol flow of ACE. Adapted from [12].



away. Operators must be given the choice to push updates to their devices, and thus servers must be able to initiate contact with devices.

3.4 Update Procedure Authorization

A flexible architecture enables different configurations of servers, operators, and devices. An operator might be authorized to update all parts of all devices, or be constrained to updating a specific piece of functionality for a subset of the devices. Operating system vendors might be allowed to push security updates for the operating system but not change the application code. Controlling access rights is a security issue and the architecture must support it.

How clients can access a protected resource through authorization tokens is described in [12] and Figure 3.2 shows an adaptation of the protocol flow from that document. The figure shows a client wanting to access a resource requesting a token from an authorization server. The token is returned to the client, which can then send a request with the token to the server holding the resource. If necessary, the resource server can send an introspection request to the authorization server asking for more information about the token in order to verify it. If everything goes as planned, the client is allowed the resource.

In the context of the update architecture, the client would be a party needing to authorize themselves, i.e. an operator or server. The access token would have to be of a lightweight variant since they are to be used on constrained devices, but which variant is implementation specific. Tokens will be sent with updates (signed manifests and images) to devices and the resource sought for is the ability to update the device. The device will thus not respond with a particular resource, but instead if the token is valid recognize the originator of the update as an authorized one and proceed with the update. Introspection is an interesting feature of ACE since it requires less from the device concerning validating tokens. Instead of doing everything on its own, the device can ask the server for additional information making the validation procedure easier. This however requires some extra communication which is power consuming for an IoT device. To use introspection or not is yet again an implementation issue, but authorization servers should be both enrolled and reachable for devices to send introspection requests if necessary.

Who is supposed to authorize? As updates will be sent from operators through servers to devices, the devices need to know the originator of an update is authorized to issue the update. The proposition is to only require operators to authorize. No matter how updates are initiated or distributed (pull vs push and manifest and image together vs separately), updates at some point must be crafted, signed, and sent by an operator. The operator is making the decisions about updating, and thus must authorize these decisions through tokens. Servers deliver the updates to devices, but do so as a relay for operators as well as acting as repositories for images and device profiles. The messages forwarded by servers will contain the authorization token granted to an operator. This token can in addition to authorizing the update on a device be used by the server to authorize the usage of the server by the operator.

Not requiring servers to authorize should not pose any issues as the update must always be prepared in a previous step by an operator that authorizes. Also, it makes the flow of applying an update easier by not needing and update server to communicate with an authorization server. Requiring update servers to authorize would not add any security to the devices as the update already will be shipped with a token authorizing the update.

3.5 Update Handling for Local Upgrades

When an update has finally arrived to the device, it must be processed and then installed. The process of decrypting, verifying, and installing the image is heavily implementation specific with most of the details being out of scope for the architecture. However, since devices operate using different hardware and bootloaders they must be given the freedom to update in the way that makes most sense for them and the architecture should support this while requiring the approach is secure.

By including optional fields such as decryption instructions, processing steps, and postconditions in the manifest, update handlers and bootloaders can take care of updates in various ways. This enables different ways of applying updates within the same architecture. What the update handlers and bootloaders of all devices have in common is that they must trust the source of the update, they must be able to decrypt the update, they must be able to verify the validity of the update, and they must be able to do this safely such that an unexpected power cycle does not brick the entire device.

Still being in the realm of very constrained devices, the bootloader should be as simple as possible. This is also a requirement stated by SUIT on the architecture. By decrypting and verifying the image at every boot the boot procedure will be secure but quite complex. Also if a power cycle occurs during image decryption, the device might not be able to recover. Decrypting the image as part of the update mechanism and writing it unencrypted to flash would allow for a simpler bootloader, but is less secure as the device would boot from an unencrypted image.

By storing the manifest containing the image digest, unencrypted images can be verified by comparing their digest to the one in the manifest. Calculating a digest is less demanding than decryption, and a power cycle would just interrupt the digest calculation instead of the decryption of the actual bootable image. In order to make sure the manifest is correct upon boot, the digest of the manifest can be calculated and stored alongside the manifest and image. Optionally the manifest can be encrypted instead as it typically will be much smaller than the image, but again a power cycle during decryption could lead to undefined behavior.

In addition to storing the manifest for verification upon boot, the manifest also has to be stored in order to prevent rollback attacks. It contains monotonically increasing sequence numbers to ensure devices do not install older, possibly vulnerable images, but in order to compare sequence numbers the most recent manifest has to be kept around. Manifests should be kept for the update handler to compare sequence numbers, and for the bootloader to verify images.

3.6 Use Cases and Example Topologies

As discussed, operators, servers, and devices can interact in many ways. The architecture tries to be flexible allowing for different network topologies and configurations. The important parts are that devices can enroll and register, all actors have valid certificates, and that updates can be authorized. The following examples all assume devices are in the maintenance stage of the life cycle, and re-enrollment and updating profiles are omitted to make the figures clearer.

Figure 3.3 shows an example architecture with one operator, one update server, one authorization server, and one device. The operator is capable of running the protocols needed to communicate in the constrained network, and can thus interact directly with both the device to send manifests and server to send images. An architecture like this allows for servers to be distributed within the constrained network, possibly as more capable IoT devices.

Figure 3.4 shows an operator sending updates entirely through an update server. The update server is located outside the constrained network and acts as an proxy for the operator to reach the devices. The update is distributed from the server to two devices, where one device accepts the update and the other rejects the update since the operator was not authorized to apply it to that device.

Figure 3.5 shows a more elaborate topology. It contains one operator, one authorization server, two update servers, two devices, and a proxy. The update is distributed to both servers, each server forwarding the update to one device each. The first device accepts and applies the update. The second device requires more information about the authorization token, and thus sends an introspection request. The authorization server is located outside the constrained network, and the introspection request must go through a proxy in order to reach it.

What these three examples have in common is that the operator

Figure 3.3: An operator communicating directly with machines in the constrained network.

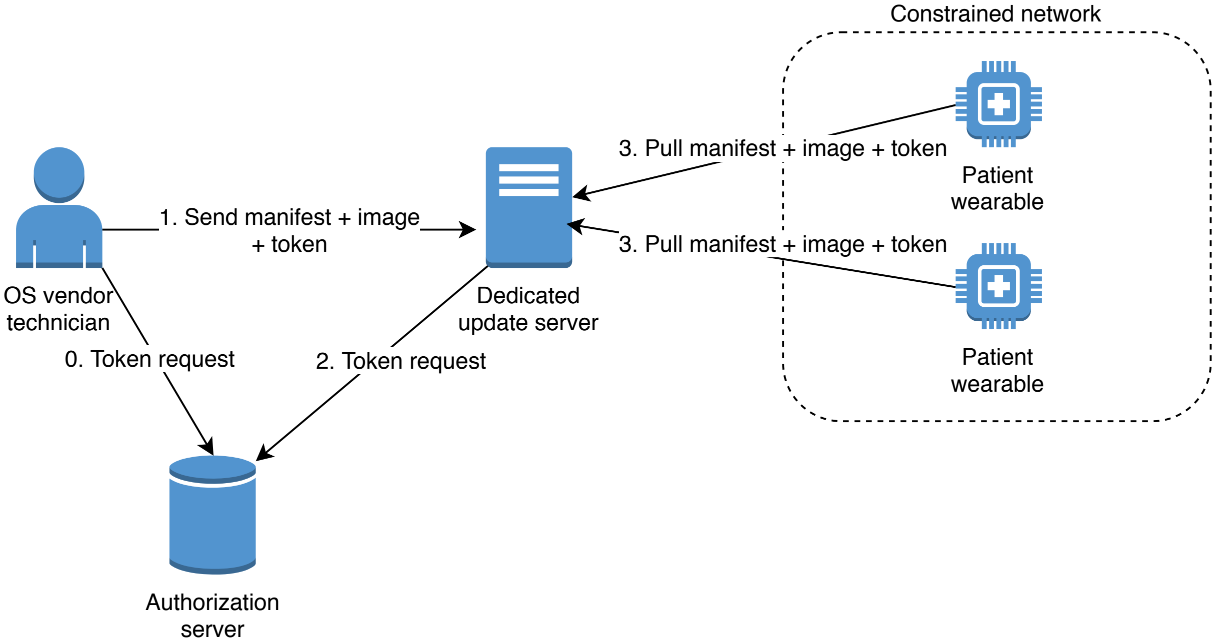


Figure 3.4: An operator communicating with an update server that pushes the update to two devices, with one device rejecting it.

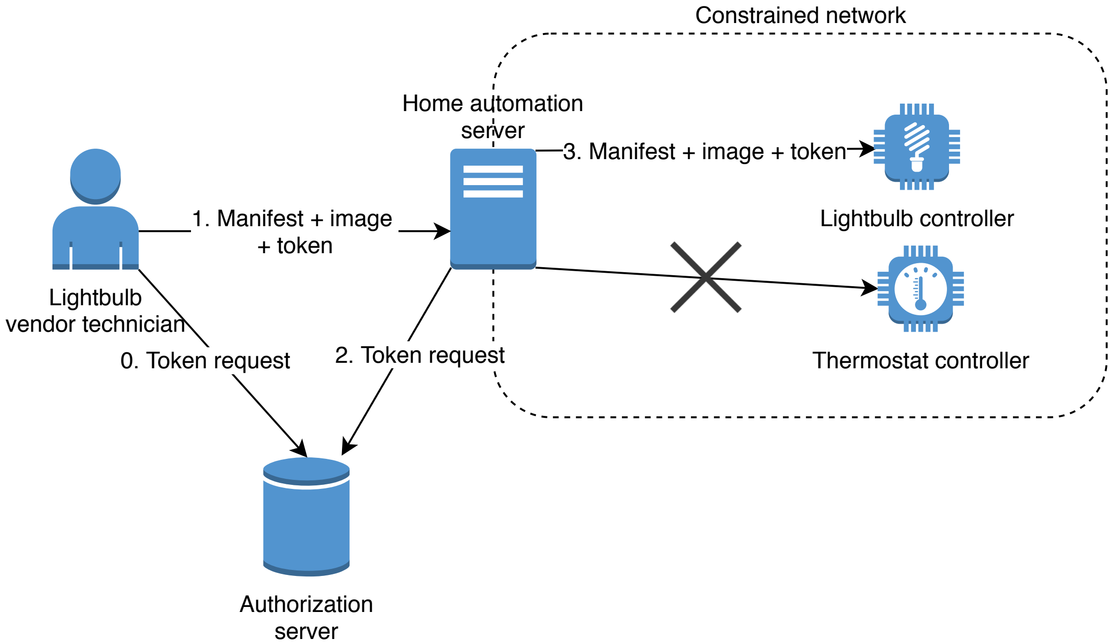
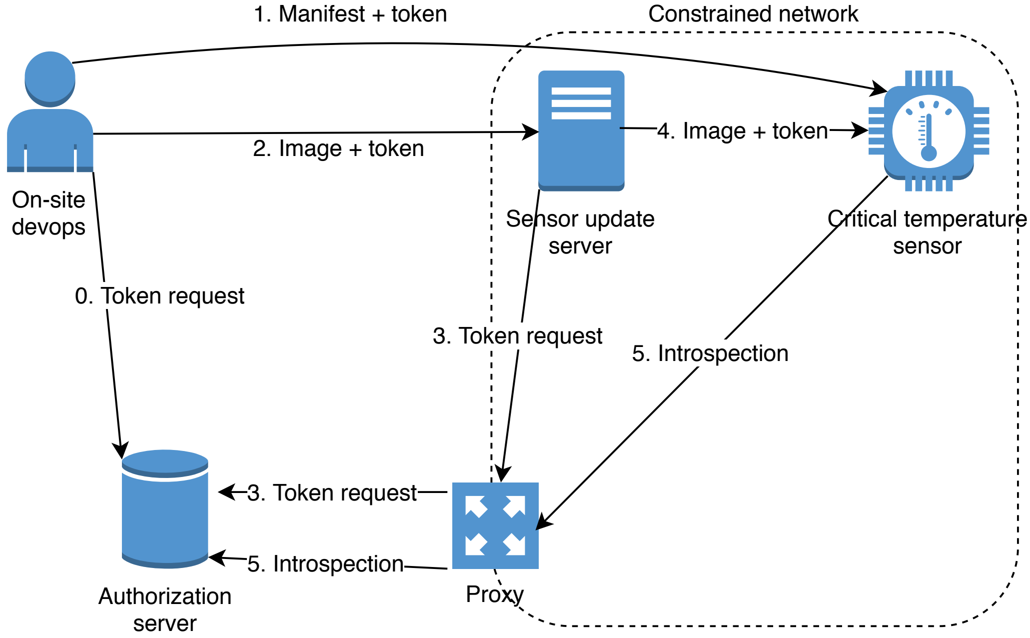


Figure 3.5: An operator pushes updates to two servers forwarding it to one device each. One device perform an introspection request to validate the authorization token.



always resides outside the constrained network, devices always reside within the constrained network, and that servers transport updates to devices alongside authorization tokens. Since operators reside in traditional networks using traditional protocols such as HTTP over TCP, a proxy mechanism may be necessary to communicate with the often UDP based constrained networks. The update server can act as a proxy, or a dedicated proxy could be used. If the operator has the proper protocols implemented, they could communicate manifests directly to a device.

Chapter 4

Internet of Things Update Life Cycle

4.1 Device Life Cycle

The five key areas discussed in sections 3.1-3.5 are all part of the life cycle of a device. From the moment a new device is deployed in a network to the point where it is taken out of action possibly several years later, the life cycle describes a holistic view of the state and operations of devices.

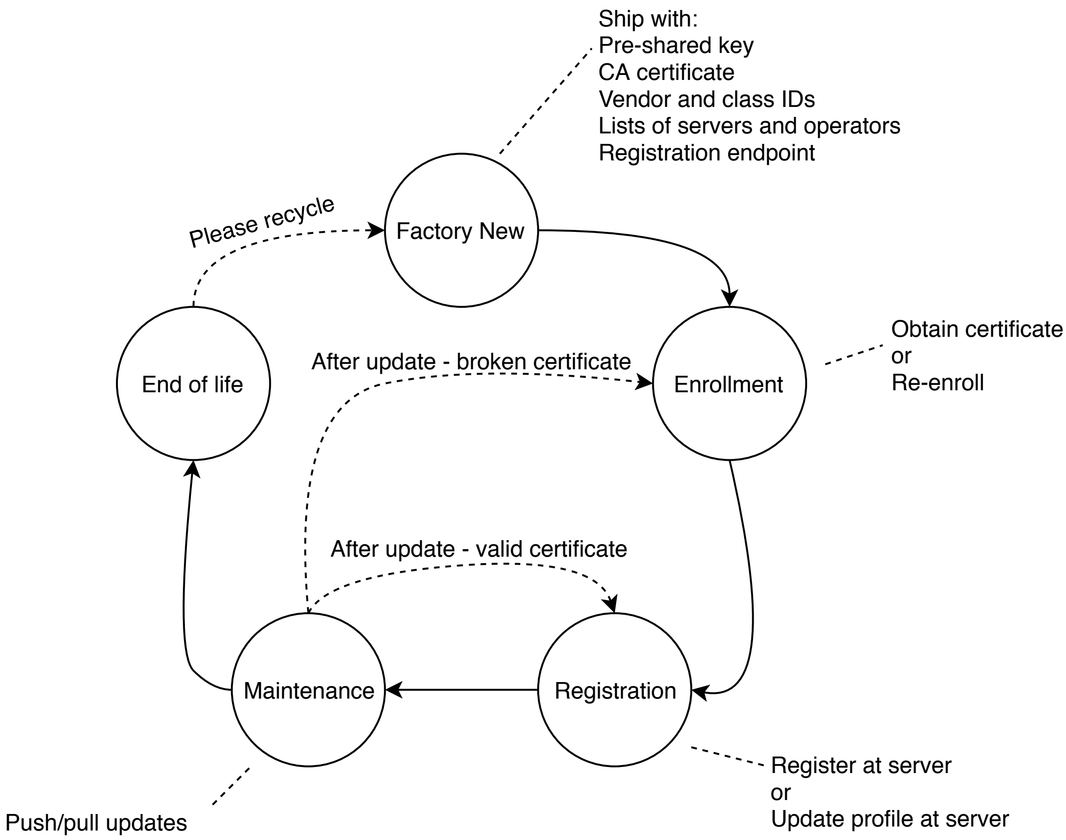
Figure 4.1 summarizes the life cycle of a device from an update perspective. The figure shows the different stages of a device from being manufactured to ending its service. There are also annotations showing what needs to be done in each stage. These stages are discussed in further detail in this section.

A factory new device that is to be installed in an IoT network needs some information in order to enroll and register at a server. A pre-shared key, CA certificate, vendor and class IDs, list of servers and operators, and and endpoint for registration is needed. These parts make sure the device can take part of the network and move to the next step of the life cycle.

Entering the enrollment stage, devices obtain certificates by enrolling at the CA. For this the pre-shared key and CA certificate is needed. After obtaining a certificate the device can be trusted by the servers and operators, as well as other devices.

The next stage is registering. The device registers at its designated servers who all create a profile for the device. The servers can now

Figure 4.1: The life cycle of a device.



reach the device, and the device is ready to be updated, and thus proceeds to the maintenance stage.

The maintenance state can be expected to last for several years, and this is where the device receives and applies updates. After an update, a device will either move back to the enrollment or registration stage. If the certificate is broken after updating, as discussed in Section 3.2, the device move back to the enrollment stage to obtain a new certificate. After obtaining a new, valid certificate it can update its profile. If the certificate is still valid after the update, there is no need to re-enroll and the device moves back to the registration stage, updates the profile at the servers. Finally the device moves to maintenance once again, and so forth. The device will remain in the maintenance stage until it either breaks or is taken out of service. If you are a manufacturer of IoT devices please consider recycling or (securely) re-using devices, starting the life cycle anew.

Chapter 5

Internet of Things Architecture Profiles

Bibliography

[1] Peter Jonsson et al. *Ericsson mobility report*. Nov. 2018. URL: https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-november-2018.pdf.

[2] Nicole Perlroth. *Hackers Used New Weapons to Disrupt Major Websites Across U.S.* Oct. 2016. URL: https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html (visited on 01/16/2019).

[3] Alex Hern. *Hacking risk leads to recall of 500,000 pacemakers due to patient death fears*. Aug. 2017. URL: https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update (visited on 01/16/2019).

[4] Syed Ali, Ann Bosche, and Frank Ford. *Cybersecurity Is the Key to Unlocking Demand in the Internet of Things*. Oct. 2018. URL: https://www.bain.com/insights/cybersecurity-is-the-key-to-unlocking-demand-in-the-internet-of-things/ (visited on 12/10/2018).

[5] *Software Updates for Internet of Things (suit)*. URL: https://datatracker.ietf.org/wg/suit/about/ (visited on 01/16/2019).

[6] Jon Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://rfc-editor.org/rfc/rfc768.txt.

[7] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347. URL: https://rfc-editor.org/rfc/rfc6347.txt.

[8] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: `10.17487/RFC7252`. URL: `https://rfc-editor.org/rfc/rfc7252.txt`.

[9] Carsten Bormann and Zach Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. Aug. 2016. DOI: `10.17487/RFC7959`. URL: `https://rfc-editor.org/rfc/rfc7959.txt`.

[10] Peter Van der Stok et al. *EST over secure CoAP (EST-coaps)*. Internet-Draft draft-ietf-ace-coap-est-07. Work in Progress. Internet Engineering Task Force, Jan. 2019. 46 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-ace-coap-est-07`.

[11] Max Pritikin, Peter E. Yee, and Dan Harkins. *Enrollment over Secure Transport*. RFC 7030. Oct. 2013. DOI: `10.17487/RFC7030`. URL: `https://rfc-editor.org/rfc/rfc7030.txt`.

[12] Ludwig Seitz et al. *Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)*. Internet-Draft draft-ietf-ace-oauth-authz-20. Work in Progress. Internet Engineering Task Force, Feb. 2019. 79 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-ace-oauth-authz-20`.

[13] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. DOI: `10.17487/RFC7228`. URL: `https://rfc-editor.org/rfc/rfc7228.txt`.

[14] Brendan Moran et al. *A Firmware Update Architecture for Internet of Things Devices*. Internet-Draft draft-ietf-suit-architecture-02. Work in Progress. Internet Engineering Task Force, Jan. 2019. 22 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-suit-architecture-02`.

[15] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. *Firmware Updates for Internet of Things Devices - An Information Model for Manifests*. Internet-Draft draft-ietf-suit-information-model-02. Work in Progress. Internet Engineering Task Force, Jan. 2019. 32 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-suit-information-model-02`.

[16] Microsoft. *The STRIDE Threat Model*. Nov. 2009. URL: `https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)` (visited on 01/17/2019).

[17] contiki-ng. *Contiki-NG*. 2019. URL: `https://github.com/contiki-ng/contiki-ng` (visited on 01/18/2019).

[18] contiki-os. *Contiki*. 2018. URL: `https://github.com/contiki-os/contiki` (visited on 01/18/2019).

[19] "IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)". In: *IEEE Std 802.15.4-2003* (2003). DOI: `10.1109/IEEESTD.2003.94389`.

[20] M. Kovatsch, S. Duquennoy, and A. Dunkels. "A Low-Power CoAP for Contiki". In: *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*. Oct. 2011, pp. 855–860. DOI: `10.1109/MASS.2011.100`.

[21] contiki-ng. *TinyDTLS*. 2018. URL: `https://github.com/contiki-ng/tinydtls` (visited on 01/18/2019).

[22] Adam Dunkels et al. "Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. Boulder, Colorado, USA: ACM, 2006, pp. 29–42. ISBN: 1-59593-343-3. DOI: `10.1145/1182807.1182811`. URL: `http://doi.acm.org/10.1145/1182807.1182811`.

[23] contiki-os. *Multithreading*. 2014. URL: `https://github.com/contiki-os/contiki/wiki/Multithreading` (visited on 01/23/2019).

[24] *Zolertia Firefly Revision A2 Internet of Things hardware development platform, for 2.4-GHz and 863-950MHz IEEE 802.15.4, 6LoWPAN and ZigBee® Applications*. ZOL-BO001-A2. Revision A2. Zolertia. Dec. 2017.